Learning Control Knowledge within an Explanation-Based Learning Framework

Roberto V. Desimone

Ph.D.

University of Edinburgh

1989

To my parents,

Stefano and Erminia

Acknowledgements

To my supervisor, Alan Bundy, I offer my sincerest thanks. His continual support and encouragement of my research has meant a great deal to me. He was always able to come up with suitable ideas and suggestions, when they were really needed the most. By his example, he has taught how to conduct academic research in a mature and efficient manner. My sincerest thanks to Alan are but a tiny repayment for the great debt I owe him. I could not have asked for a better PhD supervisor.

Many thanks to my other supervisors, Lincoln Wallen and Fausto Giunchiglia. Lincoln was always there to explain rather formal concepts. I am grateful for the patience with which he spent many hours describing the workings of NuPRL and the effort he put into generating example proofs for this thesis. I am particularly grateful for his help while I was writing the many outlines of this thesis.

Fausto became a co-supervisor during the writing up period of my thesis. His misfortune was to be afflicted with the task of reading and commenting on several drafts of my thesis. I am grateful for the considerable time and effort he put into this endeavour.

Over the past few years I have been very fortunate in pursuing my doctoral research within several stimulating environments: in the Mathematical Reasoning Group, the Department of Artifical Intelligence and, finally, in the AI Applcations Institute. Long may these environments remain stimulating and productive!

My thanks to the many characters of the Mathematical Reasoning Group, who have made my life as a PhD student very stimulating, both academically and socially: Robin Boswell, Paul Brna, Mitch Harris, Jane Hesketh, Liam Lynch, Pete Madden, Steve Owen, Mary-Angela Papalaskaris, Dave Plummer, Dave Robertson, Bernard Silver, Andrew Stevens, Mike Uschold, Toby Walsh and Richard White.

My thanks to the Science and Engineering Research Council SERC for funding my studentship, and for computing resources available under SERC grants: GR/D/44874 and GR/D/44270 and Alvey IKBS/137. My thanks also to the Commission of the European Community for funding my travels through Europe under the collaborative project on machine learning and knowledge acquisition, COST-13.

To my wife, Alison, for not complaining too much. Without her continual support and encouragement, this thesis would probably still be in the writing.

Abstract

The research reported in this thesis is concerned with extending existing techniques within machine learning for the task of learning more expressive control knowledge. In particular, it discusses research pursued within a sub-area of machine learning, called Explanation-Based Learning (EBL).

Past efforts in EBL have resulted in some success in learning *useful* control knowledge for problem solving. Much of this success derives from experience of previous research into problem solving, especially for characterising much of the domain knowledge required for learning. Thus, previous EBL programs have identified some descriptors about the state of the problem and its solution, which may provide useful knowledge for controlling the problem solving process.

The solutions described within this thesis improve the ability to learn more expressive control knowledge, by extending the power of the EBL approach. Part of the solution is achieved by integrating the best features of the previous EBL programs, in the following manner:

- identify the various types of control knowledge specified in EBL programs to date,
- analyse the learning techniques involved in these EBL programs,
- develop an extended EBL approach which incorporates the best features of these techniques and learns *all* these different types of control knowledge

This approach reflects a standard methodology of extending scientific knowledge based upon a rational reconstruction of the fruits of previous research.

As a result of the rational reconstruction within this thesis, importance has been given to the role of a *meta-language*, for describing control knowledge. The specifications of operators may be represented at the meta-level as general properties of problems and their solutions. Together with mechanisms for connecting

the specification of operators involved in the solution, generalised plans may be learned from examples of solutions to such problems.

The importance of the meta-language within the extended EBL program becomes more apparent when it is applied to the task of learning theorem proving strategies or *proof plans*. The complex nature of theorem proving strategies requires all the expressibility that is available with the extended EBL program. Hence, it provides a rich domain for exploring the role of a meta-language in learning useful control knowledge.

I declare that this thesis has been composed by myself and that the work described in it is my own.

Roberto Desimone

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The problems and their solutions

The research reported in this thesis is concerned with extending existing techniques within machine learning for the task of learning more expressive control knowledge. In particular, it discusses research pursued within a sub-area of machine learning, called Explanation-Based Learning (EBL).

Past efforts in EBL have resulted in some success in learning *useful* control knowledge for problem solving. Much of this success derives from experience of previous research into problem solving, especially for characterising much of the domain knowledge required for learning [1]. Thus, previous EBL programs have identified some descriptors about the state of the problem and its solution, which may provide useful knowledge for controlling the problem solving process.

However, in most cases, the quality of this knowledge is limited in its general description of the solution. Some programs provide *tactical* knowledge about the local application and contribution of the operators. Others represent *strategic*

[1] Such domain knowledge takes the form of the operators involved in problem solving, together with their specifications, and descriptions of the problem state.

knowledge about the operators involved in the entire solution. The generality of the control knowledge varies for each program, depending on the language used to describe the problem states.

The solutions described within this thesis improve the ability to learn more expressive control knowledge, by extending the power of the EBL approach. Part of the solution is achieved by integrating the best features of the previous EBL programs, in the following manner:

- identify the various types of control knowledge learnt in EBL programs to date,
- analyse the learning techniques involved in these EBL programs,
- develop an extended EBL approach which incorporates the best features of these techniques and learns *all* these different types of control knowledge

Importance has been given to the role of a *meta-language*, for describing control knowledge. The meta-language is distinct from the language in which problems are normally described, *viz* the object-language, by being able to represent knowledge about the object-language. In addition, it not only provides a good separation of control knowledge from factual knowledge involved in the description of the problem, but also provides a good language for describing general properties of the problem and its solution, which is an important consideration for a learning system. The specifications of operators may be represented at the meta-level as general properties of problems and their solutions. Together with mechanisms for connecting the specification of operators involved in the solution, generalised plans may be learned from examples of solutions to such problems.

The importance of the meta-language within the extended EBL program becomes more apparent when it is applied to the task of learning theorem proving strategies or *proof plans*. The complex nature of theorem proving strategies requires all the expressibility that is available with the extended EBL program.

Hence, it provides a rich domain for exploring the role of a meta-language in learning useful control knowledge.

To summarise: the contribution of this work is extending the expressive power of the EBL approach for learning control knowledge. In particular, the main contributions are

- showing how more expressive control knowledge can be learned from successful solutions to given problems by integrating the best features of previous EBL work.

- applying this extended EBL approach to a different domain. One that has not been dealt with by current EBL techniques. Also, one that requires more expressive control knowledge, in order to perform successful problem solving.

The author believes that the resulting extended EBL approach provides a significant contribution to the area of explanation-based learning research and reflects the importance of the EBL approach to the field of machine learning research.

In the rest of this introductory chapter, some details are provided which are not dealt with in the main body of the thesis. Section 1.2 discusses the motivation for learning control knowledge and for learning about proof plans. In section 1.3, details are provided about the relevance of the EBL approach for learning control knowledge. Finally, in section 1.4, the structure of the thesis is presented.

1.2 Learning control knowledge

1.2.1 Importance of control knowledge

AI researchers have always been interested in solving problems. Whether these are abstract puzzles or more practical tasks, researchers have been keen to find

clever ways of applying their problem solving operators, in order to avoid combinatorial explosion of the search space for a solution.

By specifying the kinds of problems to which an operator may be applied, the problem solving system can choose the most appropriate operator in order to achieve the required goal. The specifications of operator represent knowledge which can be used to control the application of the operator. Such control knowledge enhances the power of problem solving systems by reducing the search for the most appropriate problem solving operators.

As the problems become more and more complex, the solutions require more expressive control knowledge that describe tactics and strategies for applying sequences of operators. Eventually, it seems wasteful to throw away this control knowledge once the problem has been solved, since it could be used to solve other similar problems.

Extracting such control knowledge from successful solutions to problems and generalising it, as much as possible, to deal with many similar problems, therefore, becomes an profitable enterprise. The necessity to acquire more control knowledge becomes more evident as new problem solving operators are introduced. Thus, learning control knowledge becomes an important way of improving the performance of a problem solving system.

1.2.2 Learning about Theorem Proving

The EBL research, discussed within this thesis, involves a domain which links previous and current work in the Edinburgh Mathematical Reasoning Group [2], *viz*, that of theorem proving.

---

[2]The Edinburgh Mathematical Reasoning Group (MRG) was set up by Prof. Alan Bundy, during the 1970s. It forms part of the Department of Artificial Intelligence at the University of Edinburgh.

Such a domain has not been dealt with much by EBL researchers, particularly, for the task of learning of theorem proving strategies or proof plans. Compared with other domains in which EBL techniques have been applied, theorem proving should prove a rich domain for the learning of control knowledge. The inherent complexities of proof trees means that very expressive control knowledge is required for guiding the choice of proof operators.

The goal of the research is to learn such proof plans from examples of proof traces. An analysis of the reasons for each step in a proof and how these steps fit together to form the entire strategy would provide the system with a better understanding of proof plans. Explanation-based learning techniques emphasise the explicit representation of such domain knowledge as a network of schema structures and dependency links from which new schemata are created. These techniques are thus suited to the task of acquiring proof plans from examples of proof traces.

1.3 Explanation-Based Learning

1.3.1 Background

During the past decade, there has been increasing interest in machine learning. Small workshops that used to meet occasionally have grown in number, size and frequency, such that an international conference, dedicated to machine learning, appears to be a certain *annual* fixture [3]. Successful applications of machine

[3] Although, most machine learning papers arise from the USA, interest within Europe has also been expanding over the past few years. The main forum for discussions within Europe have been the European Working Sessions on Learning, which have taken place annually since 1986.

learning techniques for automated rule learning [Michalski 82] have resulted in a keen interest by many industrial researchers [4].

Machine learning has always been considered a stimulating, but challenging area of research, requiring a sound understanding of many general AI issues, including knowledge representation and problem solving. Research into these areas, throughout the periods of the 1950s and 60s and early 1970s, has laid the foundation for today's growing interest in machine learning.

Most machine learning research to date has involved either inductive learning – extracting a common structure from a set of examples and non-examples of a concept [Winston 75] [Michalski 83] – or learning by analogy – learning a target concept by analogy with an existing source concept [Kling 71]. This type of learning has been named Similarity-Based Learning (SBL), since it involves looking for similarities (and differences) between more than one instance of the concept.

During the 1980s, some members of the machine learning community have adopted a different approach to learning from examples. Their approach is deductive rather than inductive and makes a greater use of the knowledge in the domain of the concept to be learned [Mitchell 83b,Silver 84,DeJong 83]. This type of learning has been named Explanation-Based Learning (EBL) [5].

1.3.2 Description of EBL

EBL researchers have been exploiting experiences gained from research into the area of knowledge representation in the 1970s. Previous SBL approaches relied mostly on a pattern matching process, possibly with reference to some gen-

---

[4]Attendance at recent workshops has comprised a steadily increasing number of industrial and other non-academic researchers.

[5]The term Explanation-based learning was adopted at the Third International Machine Learning Workshop in Pennsylvania, USA, in 1985.

eralisation language, in order to determine the most general concept. Better representation of the domain knowledge now permits more inference to take place during learning and this influences the generalisation process. With more emphasis on knowledge representation, there seems to be a natural progression away from inductive learning techniques *alone*, towards an approach to learning that includes deductive techniques, such as EBL.

The EBL approach is heavily knowledge-based and requires a great deal of knowledge about the domain of the concept to be learned [6]. New concepts are learned using existing concepts and these are incrementally added to known concepts [7].

Instead of providing several instances of a concept, only a single example is required for a new concept to be learned. The idea is that a strong domain theory not only helps in describing or *explaining* the new concept, but also guides the generalisation process, such that a great deal can be learned from just a single example.

Thus, the aim of EBL techniques is to *explain* as much as possible about the example, by analysing its constituent parts, in terms of the domain descriptors, and determining how they relate to each other and link together. Only descriptors that relate to each other and link together, to form the explanation, are relevant to the concept.

---

[6]Research into EBL programs has, so far, been restricted to domains where there is a good deal of knowledge about the descriptors of the domain and how these are related within the domain. Such domains are considered to have a *strong* theory.

[7]One of the aims of the EBL approach is to permit the refinement of existing concepts, when new information about the concept from other examples enhances or contradicts the old. However, this has not yet been dealt with much by EBL researchers and is the subject of current research [Lebowitz 86,Pazzani 87]. Further discussion on this issue is postponed until chapter 8 on further work.

So far, EBL techniques for learning control knowledge have been applied to various domains, including narrative understanding [DeJong 83,Mooney 85], naive classical physics [Shavlik 85], robot manipulator learning [Segre 85], algebraic problem solving [Silver 84], symbolic integration [Mitchell 83b,Porter 84, Keller 87] and game playing [Minton 84].

1.4 Overview of Thesis

So far, the discussion in this chapter has motivated the interest in learning control knowledge within an explanation-based learning framework. It has been indicated that meta-languages have an important role to play in representing and learning such control knowledge. It has also motivated the choice of theorem proving as good domain for testing the extended EBL program.

Chapter 2 presents a survey of the research to date, within the area of EBL, for representing and learning control knowledge. The distinction is made between two ways of representing control knowledge, with object and meta-level knowledge. Various types of control knowledge are identified from the literature. The contribution of each type of control knowledge for improving problem solving is described and examples given. A survey of past efforts at learning control knowledge using EBL techniques is undertaken. Each program in the survey is presented in chronological order, to give an historic perspective of the development of EBL for this learning task. The main learning techniques involved in each program are discussed and stress is placed on identifying each different type of control knowledge learned. Finally, a comparison is made of the effectiveness of each learning program in acquiring the various types of control knowledge.

Chapter 3 introduces the domain of learning proof plans by describing it as a rich domain for learning control knowledge, because of the complexities in proving theorems. Past efforts at representing some of the strategies and tactics involved in proving theorems are described. The necessary tools for constructing proof plans are described. These include a proof environment within which the

proofs are generated, *viz* the NuPRL proof development system, and a meta-language for describing properties of the state of the formulae being proved. Finally, two example proofs are presented, together with their associated proof plans.

Chapter 4 provides a deeper analysis of one of the EBL programs described in the previous chapters. The analysis is developed from a reconstruction of the important features of the program. The chapter discusses in great detail the role of meta-level inference in the two complementary problem solving and learning programs, PRESS and LP. Relevant examples in the domain of algebraic problem solving are provided. The main learning technique, *precondition analysis*, is discussed. An abstract example is introduced to highlight the main contributions of the precondition analysis technique. This example is used in the next chapter to discuss problems with the technique and, of course, their solutions.

Chapter 5 provides one of the main research chapters. In particular, it describes Extended-EBL, a program for learning control knowledge, which forms my main contribution to EBL research. It begins with the motivation for extending the precondition analysis technique. It discusses some problems with the current state of the technique, using the abstract example from the previous chapter. The solutions to these problems involve the integration of other EBL techniques and formalisms from the general area of AI research. The gradual integration of these techniques and formalisms into the core precondition analysis technique is presented with reference to examples. The complete extended EBL technique is described on a more complex example. The benefits and limitations of this approach are analysed. Finally, the extended EBL approach is compared with other related work and some conclusions made about the overall benefit of the approach.

Chapter 6 is another main research chapter. It continues the analysis of the extended EBL technique by explaining the role of the meta-language, not only for providing more general control knowledge, but also for extending two processes involved in Extended-EBL. The problem of learning the specifications of unknown operators involved in the solutions to problems is reviewed and a

solution described which makes use of the interaction between object and meta-level knowledge. This aspect of the extended-EBL technique is contrasted with related work in the area of EBL. Finally, conclusions are presented about the relevance of this work to mainstream machine learning research.

Chapter 7 describes the application of Extended-EBL to the domain of theorem proving, in particular to the task of learning proof plans. The two example proofs provided in chapter 3 are presented to the learning program. The benefits and limitations of Extended-EBL are discussed in the context of the two proof and the resulting proof plans. Conclusions are presented about the applicability of Extended-EBL to the task of learning proof plans.

Chapter 8 makes some suggestions for further work. Chapter 9 gives some general conclusions about the thesis.

Chapter 2

Representing and Learning Control Knowledge

2.1 Introduction

Growing confidence in performing problem solving tasks has led to a resurgence of interest in machine learning, particularly in the early 1980s. A new sub-area of machine learning has developed. This sub-area, known for a while by the terms 'analytic learning' or 'single-instance learning', has since 1985 adopted the name *Explanation-Based Learning* (EBL).

Many EBL researchers have developed their learning techniques and programs, based on their own experiences with problem solving systems [Fikes 72] [Mitchell 83b,Silver 83,Silver 85,Porter 84]. The representation of the operators and their specifications has provided the necessary background and domain knowledge for learning control knowledge from the output of problem solving systems.

The aim of this chapter is to provide a survey to date in representing and learning control knowledge. An understanding of the main issues for representing control knowledge and also addressing these issues from the perspective of machine learning, provides the necessary knowledge with which to further expand the learning of control knowledge.

The chapter begins with a discussion of the representation of control knowledge. Section 2.2.1 provides an analysis of the distinction between two forms of representing control knowledge, object and meta-level knowledge, and how inference at the meta-level can improve the speed and expressiveness of problem solving systems.

Most learning programs, which support problem solving systems, have emphasised the learning of a particular type of control knowledge. As a result, different learning programs have identified and learnt different types of control knowledge. In section 2.2.2, a classification is made of the various types of control knowledge that have been learnt [1]. The various types of control knowledge are described, and examples presented.

Section 2.3 provides more details of previous learning programs, with emphasis on the types of control knowledge learned and an analysis of the learning techniques required to acquire such control knowledge.

Finally, in section 2.4, a comparison is made of the capabilities of each learning program for learning control knowledge. Some concluding remarks are made about the classification.

2.2 Representing Control Knowledge

Computers that learn how to perform problem solving tasks.

This has been the goal of many AI researchers and science fiction writers. However, although automating the process of learning about problem solving has always been considered an important area of research for AI, it has also been considered to be very challenging.

---

[1] To the best of my knowledge, no classification of control knowledge, from the perspective of machine learning, has been made before.

In the early days of AI, there was little success with developing programs that learned about problem solving tasks from examples [2]. It was felt necessary that there should be a greater understanding of how problem solving tasks could be performed by a computer, before it could attempt to learn about them.

In the 1950s and 60s, the work of Newell and Simon on LT [Newell 63] and the general problem solver (GPS) [Newell 72] showed the difficulties in attempting to achieve general theories about strategies for solving problems. As a result, in the 1960s and 70s, *representational* issues became a very hot topic of research for a while. Advances were made in the representation of problem descriptions using semantic networks [Quillian 68,Raphael 68,Schubert 76], situation calculus [McCarthy 69], propositional and predicate logic [Nilsson 80], etc. Production rules [Newell 73] and frames [Minsky 75] were used for representing problem solving operators.

Some researchers have looked at the separation of control knowledge from factual domain knowledge [Bundy 81,Davis 80,Stefik 81,Kowalski 79a]. The use of these formalisms and mechanisms for problem solving and the emphasis on representing control knowledge explicitly, has resulted in a much better understanding of the problem solving process.

The next two sections 2.2.1 and 2.2.2 provide a discussion of different forms for representing explicit control knowledge and the different types of control knowledge that have been described in the literature.

---

[2]The work of Samuels on learning about strategies for game playing, was noted for a certain amount of success in the early 1960s. However, such success was limited to these simple games, such as checkers (draughts). Extending the work to the domain of chess has proven much more difficult. Attempts to automate the learning of strategies for playing chess are still being carried out to date [Michalski 77,Good 77].

2.2.1 Representing Object and Meta-Level Knowledge

Control knowledge can be represented in the same language as the problem is described, in which case, it is said to be represented at the *object*-level. The object-level language is by definition the language in which problems are presented to the problem solver. Often operators themselves are described in object-level terms, as rewrite rules. In such cases, problem solving is thus performed entirely at the object-level.

Object-level terms provide descriptions of the actual *states* of the problem. Thus, control knowledge at the object-level basically relies on descriptions, or partial descriptions, of the problem state.

On the other hand, control knowledge can also be represented in terms of another language, at the *meta*-level. The meta-level is a description of the representation of the object-level. Thus, terms in the meta-level language provide knowledge about the relationships between terms in the object-level language. This permits a language with which one can reason about relationships between parts of the problem state. Examples of the distinction between object-level and meta-level control knowledge are described in the next section 2.2.2.

Adopting the same language, for representing knowledge about the problem states and for the necessary control knowledge, relating to the operators involved in solving the problem, leads to a simple problem solving system. However, the cost of this simplicity is a problem solving system which is messy and inefficient.

Although many problem solving operators perform similar tasks, the control knowledge that specifies them is often too specific for such relationships between operators to become evident. As a result when an operator fails to apply because it is totally inappropriate, similar operators will still be examined. Thus, many fruitless searches are performed through branches of the search space which will not produce the best operator to apply next.

This lack of generality of the control knowledge described at the object-level also leads to an inefficient problem solving system. If there are many specific operators, attempting to match the specifications of each of these operators

to the current problem state can be very expensive. Likewise, if the operator is applicable to many problem states, then the control knowledge can often comprise large conjunctions of object-level terms. In both cases, choosing the best operator to apply next, involves searching a very large space of operators, including many fruitless branches.

The separation of the factual knowledge at the object-level from control knowledge at the meta-level makes programs much more modular [3]. Control knowledge at the meta-level can provide much more general descriptions about the problem state. Thus, operators which perform similar tasks can be related by common terms in their specifications.

This constrains search in two ways:

- pruning branches in the meta-level search space removes correspondingly larger parts of the object-level search space. Thus, when an operator is found to be totally inappropriate, other similar operators are not considered,

- it avoids massive searches through a large conjunction of object-level terms, since the meta-level terms capture more generality.

The more general nature of the descriptions at the meta-level also provides a better language for describing the *strategies* involved in solving problems.

Most of the above discussion has revolved around describing control knowledge for the operators involved in problem solving tasks. The operators describe ways of transforming the current problem state into some other description of the problem state, and, hopefully, one of the operators will lead to the desired solution state. A description of the sequence of operators, which transforms the

---

[3]The separation of the representations also makes it easier to learn either kind of knowledge, factual or control. Within this thesis only the learning of control knowledge is examined.

current state of the problem into the solution or goal state, together with the control knowledge associated with each of the operators, possibly at the meta-level, provides the basis for describing a plan or strategy for solving problems or achieving sets of goals. This discussion is developed further in the rest of the thesis.

The algebraic problem solving system, PRESS [Bundy 81,Sterling 82], showed how the use of meta-level knowledge about the domain of algebra and performing inference at the meta-level provided an efficient and expressive problem solver [4]. [Davis 80] shows how meta-rules, within the TEIRESIAS program, provide a stronger notion of strategies for problem solving and reasoning about such strategies.

More discussion and examples of the differences in representing control knowledge with object and meta-level languages are given in the next section 2.2.2 and in chapter 4.

2.2.2 Types of control knowledge

The previous section has compared the use of object and meta-level languages for describing control knowledge. This section discusses the various *types* of control knowledge that can be used within a problem solving system. The following is a list of different types of control knowledge that have been used in the areas of machine learning and planning:

- applicability of plans and their operators
- contribution of operators within their plans
- explicit structure of plans

---

[4]More details about PRESS and the role of meta-level knowledge and inference, are given later on in this chapter and the next.

- cost/benefit analysis for learning plans

Most of the programs, discussed later in this chapter, characterise one, or, at most two, of the above types of control knowledge. My claim is that an *effective* and efficient learning and/or problem solving system must be able to represent all of the above types and possibly others.

The discussion in the next sections explain what each type of control knowledge does and how useful it is. The explanation is expanded to describe the forms it may take and examples are given.

Applicability of plans and their operators

Deciding *when* to apply an operator or a plan has been a very important consideration for problem solving systems. The applicability of plans and their operators provides a vital component of control knowledge [5]. Certainly, most problem solvers characterise this type of control knowledge as part of the *specifications* of the known operators and plans.

Many recognise this type of control knowledge in the form of *preconditions* to the operators or plans. Preconditions represent descriptions about the state of the problem which are considered to be *necessary* for the application of the plan or operator, so as to solve a problem which matches them.

Such descriptions have been characterised either at the object or meta-level [6]. Preconditions, described in object-level terms, either denote the entire state of the problem or a partial representation of the state, often as the left-hand side of rewrite rules or the antecedent of production rules.

---

[5]Some researchers would have us believe it to be the only component of control knowledge, particularly within the machine learning community.

[6][van Harmelen 87] distinguishes three situations: purely object-level rules, meta-level conditions mixed with object-level rules and separate object and meta-levels.

For instance, the following algebraic equation involves the quadratic expression

$$2x + x^2 + 3x + 6 = 0 \tag{2.1}$$

Before the equation can be factorised, it must be tidied up by collecting together all like terms. In this case, by combining the two terms containing the unknown variable x, *ie* $2x$ and $3x$.

The following rewrite rule may be applied to collect all like terms in equation 2.1

$$Ax + Bx \Rightarrow (A + B)x$$

The application of this rewrite rule could be determined by matching the left-hand side of the rule to the above equation. In this case, the left-hand side represents factual knowledge at the object-level. Explicit control knowledge for applying this rewrite rule could be represented at the object-level by preconditions, such as $A \neq -B$. This would restrict the application of the above rewrite rule which results in the right hand side of the rule rewriting to $0x$

On the other hand, the *meta-level* language provides a description of the *properties* of the state of the problem, rather than the descriptions of the states themselves. The meta-level language can often provide terms which are much more *general* than their object-level counterparts, since it can describe general properties of many states of the problem. Thus, preconditions represented by meta-level terms can provide a more general description of the type of problem to which the plan or operator may be applied.

Referring again to the previous example, equation 2.1, the next step involves the operation of collecting like terms. The meta-level preconditions for this operation can be represented by some property, such as the multiple occurrence of the unknown variable. This could be denoted by the predicate, $mult\_occ(X, Eqn)$, where X is the unknown and Eqn is the equation containing the unknown variables. This precondition can match many more problem states than is possible with the previous object-level precondition based on the left-hand side of a

rewrite rule. Chapter 4 provides more examples of meta-level preconditions and their use to control the application of sets of rewrite rules.

Matching Preconditions When it comes to matching the preconditions to some problem state, the choice of one or other of these representations, object or meta-level, does have a significant effect on the quantity of processing required. Matching object-level preconditions to a problem state involves a pattern matching process. For each precondition an attempt is made to unify it with some part of the object-level description of the problem state. Thus, representing preconditions in object-level terms has been favoured by many, because simple pattern matching can be performed very efficiently.

In the case of the above example, matching the preconditions involves unifying Ax and Bx with the equation 2.1. The following substitutions, $\{A/2, B/3\}$, or vice-versa, are obtained.

However, matching preconditions described in meta-level terms requires more than a simple unification of terms. An inference is required, in order to check whether certain properties exist in the problem state. Although, the matching process may be more costly and time consuming, the generality of the preconditions means that the plan or operator may be applied more effectively and for many problems states. Hence, the cost of the matching and the benefit of having more general preconditions may produce a net positive benefit.

Thus, for the example above, matching the precondition, $mult\_occ(X, Eqn)$, involves inferring that the property exists in the problem state. However, this property can represent any number of terms containing the unknown, X.

This issue is expanded further in chapter 6.

Preconditions of plans and operators The discussion has focussed so far on the form of the preconditions and how this affects the matching process. Another important point concerns the distinction between preconditions for an operator and those for a plan or sequence of operators. The former are provided

in most problem solving systems, the latter are not. After the application of each operator in the sequence, there is always another sequence of operators remaining until the last operator is reached. Thus, representing the preconditions for each stage of the plan means providing preconditions for each of these sequences of operators in the plan.

Matching the current description of the problem state with the preconditions for the next stage provides more heuristic knowledge about whether the application plan should succeed, although this is not guaranteed.

This issue is examined further in chapters 5 and 6.

Contributions of operators within plans

Representing the contribution of operators within plans provides an important type of control knowledge that is used widely in the area of planning [7]. In particular, it is considered vital information for controlling the execution of plans and for aiding the repair of faulty plans. Knowledge about what each part of the plan achieves can provide a reason for the role of each operator. This information is important when monitoring the execution of a plan for determining whether the application of an operator has been successful. The application of an operator can be considered successful, if it achieves the purpose for which it was intended within that particular plan. Thus, it provides a means of checking whether the problem solving is proceeding according to plan.

However, sometimes strictly following the plan does not lead to a solution to a problem. It is possible that the previously learnt plan may have been too specific, having been learnt from an example that dealt with a special case. Hence, some of the operators within the plan may need to be replaced or expanded. Nevertheless, the main structure of the plan may still be sound, it just needs *patching* at certain steps.

---

[7] The contribution of operators within a plan is often called the *teleology* of a plan or the *plan rationale.*

In this case, knowledge about the purpose of each part of the plan, provides the right sort of information for patching the plan. This may involve either choosing another operator, to add to an existing operator, in order to achieve the purpose of the step, or else replacing the operator entirely.

The form of this knowledge has been represented by either the *postconditions* or *effects* of each operator. In this thesis, I make the distinction between these two forms. The postconditions denote properties of the state of the problem that *should* exist for the operator application to be deemed successful. The effects provide information about the state transition *caused* by the operator. Thus, the postcondition only refers to the state after the application of the operator, and the effects refer to the relationship between the states before and after the application.

The distinction between these two forms can be shown more clearly, by continuing on from the example in the previous section 2.2.2. The following equation

$$x^2 + 5x + 6 = 0 \tag{2.2}$$

describes the state of the problem before the *factorisation* operator is applied. The resulting equation after factorisation is represented by the following expression

$$(x + 2)(x + 3) = 0 \tag{2.3}$$

The postconditions of the factorisation operator could be represented by various properties of the resulting problem state in equation 2.3, *ie* that the terms on the left hand side of the equation comprise a 'product of sums' and that it contains the factors, $(x+2)$ and $(x+3)$. These could be denoted by the following predicates,

$$product\_of\_sum(Eqn2)$$
$$list\_of\_factors(Eqn2, Factors)$$

where $Eqn2$ denotes the equation 2.3 and $Factors$ denotes the factors, $(x + 2)$ and $(x + 3)$.

The effects of the factorisation could be represented by the following relationship between the two equations 2.2 and 2.3,

$$factorise(Eqn1, Eqn2, Factors)$$

where $Eqn1$ refers to equation 2.2, $Eqn2$ refers to equation 2.3 and $Factors$ to the factors, $(x + 2)$ and $(x + 3)$.

This discussion about the distinction between postconditions and effects is continued in chapters 3 and 6.

Explicit structure of plans

One of the most essential components of a plan is the structure of its operators. The structure contains knowledge about which operator to apply next to the current state of the problem. Without representing the structure of its operators, there is no clear representation of the strategy involved in the plan [8].

To date, two main ways have been adopted for representing the structure of a plan

- the ordering of its operators
- the inter-connectivity between its operators

The ordering of the operators can be described in an *implicit* or *explicit* manner. The implicit description of the ordering is represented by the sequence of operators, eg by a simple linear list or by a finite state machine. The emphasis is on the position of the operator in the sequence to provide the information about what comes next.

---

[8]Some would go further and say that a plan consists purely of the structure of its operators alone. But, this neglects the other types of control knowledge, discussed within this section.

The explicit description is generally provided in the form of an ordering relation between operators [9]. The ordering relation provides constraints on the positioning of operators in the sequence [10]. Such constraints are often required to avoid undoing of effects of previous operators or getting into unproductive loops that lead to unsuccessful paths.

The structure of a plan may also be described in terms of how its operators are interconnected. Such information complements the knowledge provided by the contribution of operators described in the previous section 2.2.2. Whereas the postconditions or effects of an operator tell us *what* an operator achieves, the interconnections provide us with knowledge of *how* this achievement helps the application of other operators within the plan.

Thus, the interconnections of the operators are represented by links between the postconditions or effects of one operator and the preconditions of another. These links may occur between adjacent operators, showing how the effects of one operator help the application of the next operator in the sequence. Alternatively, these links may span several operators. Such a situation arises where an operator produces postconditions which are useful for an operator later on in the sequence, beyond the next operator [11].

Cost/Benefit analysis for learning plans

Another type of control knowledge, which has only recently been considered within the EBL community, involves the use of evaluation criteria, both within

---

[9]Explicit ordering relations are becoming more popular as temporal reasoning is required for problem solving.

[10]The ordering of operators may *partial* or *total.*

[11]In the area of planning, such postconditions are called *spanning conditions*, since they span more than one operator in the sequence.

the learning procedure or as a post-process after learning [12]. So far, evaluation criteria have been used for two purposes.

- for deciding when to stop the learning process.
- for deciding how *useful* an operator or plan is.

The problem of deciding *when* to stop learning about a particular operator or plan, has always been avoided in machine learning. It is very difficult to determine when enough has been learnt, and yet one would not want the system to continue learning when there is no further benefit.

The other problem of measuring the utility of a learnt operator or plan, in some way, is just as difficult. Deciding whether its use will speed up the problem solving process involves a tradeoff between the cost of applying the learnt operator or plan, possibly in terms of processing time and/or storage space, and the benefit it produces.

Research into the use of evaluation criteria for learning is still at an early stage, so that it is unclear what form they should take. For both of the purposes described above, the aim is to measure improvement in performance. In one case, learning should stop when sufficient improvement in performance has been achieved, and, in the other case, this measure of improvement should be recorded. How does one determine the cut-off point for sufficient improvement? What measures of improvement are adequate?

These issues are not dealt with much in the rest of the thesis, but are referred to again later on in this chapter and in chapter 8 on further work.

---

[12] However, this topic has been addressed within the area of machine learning by [Good 77] in the early 1970s.

2.3 Learning Control Knowledge

In this section, various learning programs are presented in a chronological order. For each program, a description is given of where it was developed and by whom; the domain in which it was applied and whether it derived from previous experiences with problem solving systems. Details are given of the main learning techniques involved and the particular types of control knowledge that are stressed by each program.

Many EBL researchers have developed their learning techniques and programs, based on their own experiences with problem solving systems [Fikes 72] [Mitchell 83b,Silver 83,Silver 85,Porter 84]. The representation of the operators and their specifications has provided the necessary background and domain knowledge for learning control knowledge from the output of the problem solving system.

2.3.1 MACROPS

The first example of EBL research can be traced well before the term was coined in 1985.

In the early 1970s, a group at SRI developed STRIPS, a robot problem solving system and PLANEX, a plan execution monitoring system for use on their robot SHAKEY [Fikes 71,Fikes 72]. Having succeeded in getting SHAKEY to generate and execute plans for solving simple blocks world tasks, they went one step further and produced one of the first programs MACROPS that learned from its own planning experiences. The MACROPS learning program enables their robot to generalise and save a solution to a particular problem. Thus, a generalised plan can be used as a *macro action* for further problem solving [13]. In addition,

[13]The following description of MACROPS is taken from [Fikes 72]

recording the reasons for each operator in the plan provides flexibility in plan execution, since unsuccessful portions of the plan may be repaired or re-planned.

The plans are stored in a tabular format, called a *Triangle Table.* Triangle tables provide a format

- for storing the macro, so as to make any of its sub-sequences available to STRIPS
- for identifying the role of each operator

Figure 2–1 shows an example triangle table.

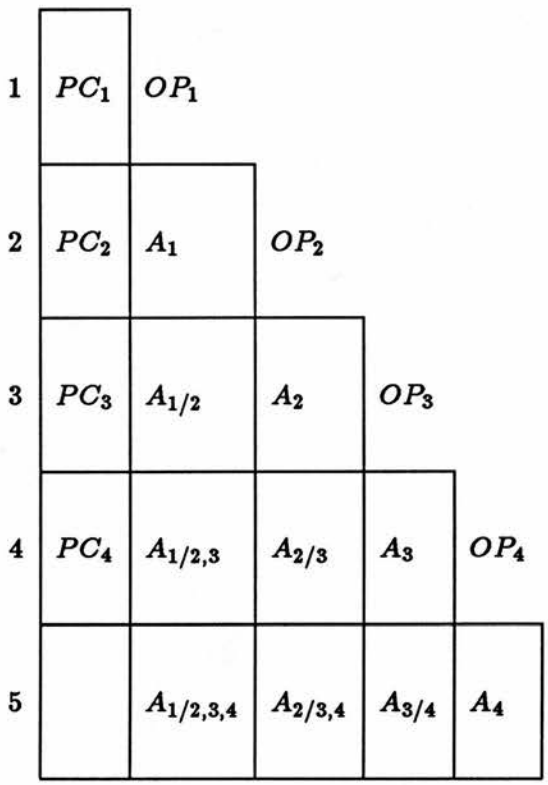
The learning procedure in MACROPS involves the following processes, which are described in the context of a plan for moving a box, $BOX1$, from room, $R2$ to $R1$, through a doorway, $D1$.

Given a successful sequence of actions for solving a problem

- store the sequence of operators, together with their preconditions, add and delete lists in a triangle table. Figure 2–2 also represents the triangle table for this specific plan.
- bring the table to its most general form, by replacing all constants in the clauses by distinct parameters, *eg* the constants $ROBOT$ and $R1$ are replaced by the parameters, $p1$ and $p2$. Figure 2–3 shows the resulting table.
- apply constraints to the parameters of each clause, by resolving them against the preconditions of each operator,

---

..the generalization involved becomes a powerful form of learning that can reduce planning time for similar tasks, as well as the formation of much longer plans, previously beyond the combinatoric capabilities of STRIPS.



A_i – add list from OP_i

$A_{i/j}$ – add list from OP_i after deletions from OP_j

PC_i – support clauses from initial model

Figure 2–1: Representation of a triangle table

	*INROOM (ROBOT,R1)		
1	*CONNECTS(D1,R1,R2)	GOTHRU(D1,R1,R2)	
	*INROOM (BOX1,R2)		
	*CONNECTS(D1,R1,R2)	*INROOM(ROBOT,R2)	
2	*CONNECTS(x,y,z)→		
	CONNECTS(x,y,z)		PUSHTHRU(BOX1,D1,R2,R1)
			INROOM(ROBOT,R1)
3			INROOM(BOX1,R1)

Figure 2–2: Triangle table for the specific plan.

	*INROOM (p1,p2)		
1	*CONNECTS(p3,p4,p5)	GOTHRU(p11,p12,p13)	
	*INROOM (p6,p7)		
	*CONNECTS(p8,p9,p10)	*INROOM(ROBOT,p13)	
2	*CONNECTS(x,y,z)→		
	CONNECTS(x,y,z)		PUSHTHRU(p14,p15,p16,p17)
			INROOM(ROBOT,p17)
3			INROOM(p14,p17)

Figure 2–3: Over-generalised plan in triangle table

eg the constraints, $INROOM(p1, p2)$ and $CONNECTS(p3, p4, p5)$ resolved together with the preconditions, $INROOM(ROBOT, p12)$ and $CONNECTS(p11, p12, p13)$, of the operator, $GOTHRU(p11, p12, p13)$, result in the following substitutions, $ROBOT \rightarrow p1$, $p2 \rightarrow p12$, $p3 \rightarrow p11$, $p2 \rightarrow p4$, $p5 \rightarrow p13$. Figure 2–4 shows the final form of the triangle table.

Note that in the generalised plan the robot is initially located in room, $p2$, and ends up in room, $p9$. The two rooms are different, with the constraint they are adjacent to the initial room, $p5$, of the box.

	*INROOM (ROBOT,p2)		
1	*CONNECTS(p3,p2,p5)	GOTHRU(p3,p2,p5)	
	*INROOM (p6,p5)		
	*CONNECTS(p8,p9,p5)	*INROOM(ROBOT,p5)	
2	*CONNECTS(x,y,z)→		
	CONNECTS(x,y,z)		PUSHTHRU(p6,p8,p5,p9)
			INROOM(ROBOT,p9)
3			INROOM(p6,p9)

Figure 2–4: Final form of generalised plan in triangle table

MACROPS learns three of the classified types of control knowledge

- explicit structure of plans
- contribution of operators within their plans
- applicability of the plan and its operators

and they are all contained within the triangle table. The triangle table is able to represent the internal structure of the plan and the contribution of each operator to this structure. The structure is represented by the interconnectivity of the rows and columns of the triangle table and the ordering of the operators. The add lists of one operator and the remainder of the add lists of previous operators become the preconditions of the next operators. The tabular format is very good for showing the contribution and interconnectivity of the operators. However, the tabular format provides a major restriction on the ordering of the operators. It is fine for a linear ordering, but it is unable to represent a tree structure of operators, as would be required for a goal reduction problem.

The applicability of the plan and its operators is represented in the triangle table by the preconditions for each operator, denoted by the marked clauses, *ie* *. If one considers the preconditions of the operators as providing knowledge about the *tactics* involved in the plan and the preconditions of the plan itself as knowledge about the *strategy* of the plan, then the triangle table is good at representing tactics. However, it is not really that good at representing the strategy of the plan. It is able to represent conditions that are required in the initial problem description for operators in the plan, but there is no representation of the preconditions for each stage of the plan.

The major contribution of MACROPS is in learning generalised plans from examples of solutions to certain problem solving tasks. This is achieved by representing the sequence of operators involved in the solution and generalising their specifications in order to capture the control knowledge described above.

Unfortunately, this pioneering work was not continued. Research into machine learning continued to develop from the mid-1970s, but focussed mostly

on inductive concept learning. However, in the early 1980s, two notable pieces of learning research, both developed from experiences with problem solving systems, resulted in techniques that emphasised different forms of control knowledge [Mitchell 83b,Silver 85].

2.3.2 LEX2

Mitchell and Utgoff at Rutgers University continued their previous research in the domain of symbolic integration, which resulted in the learning program LEX [Mitchell 81,Mitchell 83a].

In their program, LEX2, they have developed a technique called *Constraint Back Propagation* which examines the solutions to symbolic integration problems [Mitchell 83b,Mitchell 82b,Utgoff 84]. The aim of this technique is to determine the most general state of a problem such that the application of a given sequence of operators transforms the problem into a goal state.

The control knowledge learnt by this technique represents the *weakest* preconditions for a particular sequence of operators given a single example of a successful application of the sequence. This idea is by no means novel and has been used by researchers in other areas of AI research [Dijkstra 76,Waldinger 77, Follett 84].

The technique is derived from previous work in the area of program derivation, leading to the technique for finding the *weakest preconditions* for a program [Dijkstra 76]. It is also related to work in the planning work in the form of the *goal regression* technique [Waldinger 77]. Other work on program synthesis also resulted in a very similar approach to constraint back propagation, making use of *passback pairs* [Follett 84]. However, its use and success within LEX2, set it up as an important EBL technique for many years [14].

---

[14]The technique also provides another role, which addresses the issue of what has been termed, adjusting the *bias* of the description language [Utgoff 84]. The vocabulary

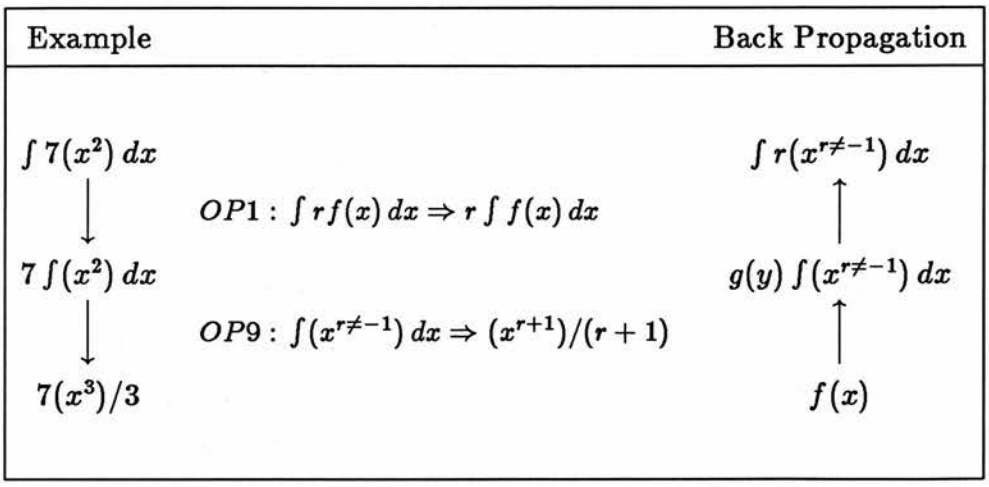


Figure 2–5: An example of constraint back-propagation

The Constraint Back Propagation technique relies on a single example of a solution to be presented to LEX2 and knowledge about the operators involved in the solution.

Firstly, an explanation tree is constructed, which represents statements about problem states and operators in the given solution tree. It explains how the sequence of operators in the current example represents a positive instance, *PosInst*, of a heuristic for transforming the problem state into a goal state. The constraint back propagation technique contributes to the process of generalising the explanation tree, by providing constraints on the description of the problem states, in order to avoid *over-generality*. Figure 2–5 shows an example of constraint back propagation applied to a symbolic integration problem.

The explanation tree for the above example in figure 2–5 results in the following definition of *PosInst*:

---

of the learning system can be extended by creating new terms, based on the weakest preconditions. These can then assimilated into the hierarchical descriptive language. These terms are then available for use in the construction of heuristics for the application of mathematical operators.

$\text{PosInst}(op1, \text{State1}) \Leftarrow (\neg\text{Goal}(\text{State1}) \wedge \text{Goal}(\text{Apply}(op9, \text{Apply}(op1, \text{State1}))))$

Generalising this explanation tree involves the following processes

- Extracting a sufficient condition for satisfying PosInst,

  $\forall s\ \text{PosInst}(op1, s) \Leftarrow (\neg\text{Goal}(s) \wedge \text{Goal}(\text{Apply}(op9, \text{Apply}(op1, s))))$

- Restating this sufficient condition in terms of the generalisation language, as restrictions on various problem states involved in the solution tree,

  $\forall s\ \text{PosInst}(op1, s) \Leftarrow (\text{Match}(\int f(x)dx, s) \wedge \text{Match}(f(x), \text{Apply}(op9, \text{Apply}(op1, s))))$

- Propagating the restrictions on various problem states through the solution tree to determine equivalent conditions on the problem state.

  $\forall s\ \text{PosInst}(op1, s) \Leftarrow (\text{Match}(\int f(x)dx, s) \wedge \text{Match}(\int r(x^{r \neq -1})dx, s))$

LEX2 performs one task very well. It learns about the applicability not only of each operator, but also of each of the stages within the plan. By this I mean that, the back propagation techniques provides the weakest preconditions for all the totally-ordered sequences and sub-sequences of operators involved in the explanation tree.

For the example in figure 2–5, there are only two stages for the plan, since there are only two operators involved in the sequence. The preconditions for each stage are derived from $\int r(x^{r \neq -1})dx$ for the sequence $op1, op9$; and $f(y) \int (x^{r \neq -1})dx$ for the trivial sequence $op9$.

Thus, the ability to represent the most general state of the problem to which the sequence of operators may be applied provides knowledge about the strategy involved in the solution, which complements the inadequacies of the MACROPS work, in this respect. Unfortunately, because the preconditions and postconditions for the operators are described at the object-level, the general applicability of the plan is limited.

Other disadvantages of the technique, as a whole, are the following.

- There is no explicit representation of the internal structure of the plan, except for the sequence of operators,

- As a result of this, and also because neither postconditions nor effects of each operator are represented, there is no explicit description of the contribution of the operators within the plan.

The next section discusses another EBL technique which was developed at about the same time as the constraint back propagation technique.

2.3.3 LP

Silver at Edinburgh University has built on previous research in the domain of algebraic equation solving and meta-level inference, which resulted in the program, PRESS [Bundy 81,Sterling 82,Silver 83,Silver 84,Silver 85]. His program LP [15] improves the performance of the PRESS algebraic problem solving system by learning from worked examples provided by a teacher. Its aim is to learn two types of information

- specifications of missing operators, called *methods*
- strategies for solving the worked examples, called *schema methods*

Methods are meant to reflect the algebraic methods that might be used by mathematicians in solving algebraic equations. These might include operations such as *factorisation*, *isolation* of the unknown variable, or *collection* of like terms. These methods comprise sets of rewrite rules which are applicable to problem states that have common properties.

Schema methods represent the strategies involved in solving the worked examples, presented to LP. They comprise the sequence of methods involved in the solution, together with the reasons for applying each method at each step of the solution. The schema method, thus, describes a plan for achieving a solution to the algebraic equation.

---

[15] The name LP is derived from Learning PRESS

The learning task in LP is performed by the Precondition Analysis technique, which analyses solutions to equation solving problems in terms of a meta-level language [16].

LP first identifies the occurrence of known algebraic methods in a worked example. Figure 2–6 shows a worked example, taken from [Silver 84,Silver 85], together with its associated methods.

Then, the precondition analysis technique analyses the solutions and determines two pieces of knowledge for each method

- Major Effects (ME)
- Satisfied Preconditions (S)

The major effects of a method represent meta-level conditions which are *necessary* for the next method in the sequence to apply. The satisfied preconditions represent those meta-level conditions which must be maintained by the current method, such that the next method applies.

What do we gain from having the major effects and satisfied preconditions for each method?

- They provide a mechanism for learning new methods when none of the known methods account for a step in the worked example. This is achieved by determining the preconditions and postconditions of missing methods.
- They provide LP with reasons for applying particular methods in the worked example, by describing how one method contributes to the next

---

[16]The worked example is presented to LP in an object-level language, *ie* comprising algebraic terms, which describe the state of the problem, as the various stages of the solution. The meta-level terms describe *properties* of the equation which it uses heavily in controlling the search for solutions and hence for identifying the algebraic methods, such as *collection, isolation* and *factorisation*, involved in worked examples.

$$\cos(x) + 2 \cdot \cos(2 \cdot x) + \cos(3 \cdot x) = 0$$

(Cosine Rule)

$$2 \cdot \cos(2 \cdot x) \cdot \cos(x) + 2 \cdot \cos(2 \cdot x) = 0$$

(Factorisation Preparation)

$$2 \cdot \cos(2 \cdot x) \cdot (\cos(x) + 1) = 0$$

(Factorisation)

$$\cos(2 \cdot x) = 0 \vee \cos(x) + 1 = 0$$

Solve first factor	*Solve next factor*
$\cos(2 \cdot x) = 0$	$\cos(x) + 1 = 0$
(Isolation)	*(Isolation)*
$x = 90 \cdot n_1 + 45$	$x = 180 \cdot (2 \cdot n_2 + 1)$

Figure 2–6: Worked Example after identification of methods

method in the sequence. Such knowledge is useful for constructing the schema method and for applying it in problem solving, particularly when the sequence of methods has to be *patched*, in order to achieve a solution.

Figure 2–7 shows the resulting schema method for the worked example shown in the previous figure, which includes the major effects and satisfied preconditions for each method.

LP stresses the learning of the contribution and applicability of each operator. Because of its representation of preconditions and postconditions at the meta-level, the control knowledge learnt is much more general than any of the other EBL techniques.

Unfortunately, the precondition analysis technique does not go far enough in providing useful strategic knowledge about the solution. Whereas MACROPS learns about the overall contribution of each operator within the plan, LP only determines the local effect of each operator, *ie* its effect on the next operator in the sequence. Thus, the internal structure of the operators involved in the solution is not as complete as that for MACROPS. This is noticed especially when some operators contribute to the applicability of other operators which are not next in the sequence.

However, this local knowledge is found to be useful, when the full sequence of operators is not known, for determining the specifications of the missing operators, *ie* it's preconditions and postconditions. Other EBL techniques require that all the operators involved in the solution are known to the learning system. Instead, precondition analysis allows the gaps in the solution to be filled.

2.3.4 PET

The next program to be discussed is again derived from previous work on a problem solving system. In their program, PET, Porter & Kibler at the University of California, Irvine have developed a system that incrementally learns heuristics for operator sequences in the domain of symbolic integration [Porter 84].

Name	Satisfied Preconditions	Major Effects
New Method (Cosine Rule)	rhs-zero(Eqn1), lhs-sum(X,Eqn1), multiple-occ(X,Eqn1)	common-subterms(X,Eqn2)
Factorisation Preparation	rhs-zero(Eqn1), lhs-sum(X,Eqn1), multiple-occ(X,Eqn1) common-subterms(X,Eqn1)	lhs-product(X,Eqn2)
Factorisation	rhs-zero(Eqn1), lhs-product(X,Eqn1) multiple-occ(X,Eqn1)	
Isolation	single-occ(X,Eqn1)	(Solution)
Isolation	single-occ(X,Eqn2)	(Solution)
Generating Equation: $\cos(x) + 2 \cdot \cos(2 \cdot x) + \cos(3 \cdot x) = 0$ Unknown: x		

Figure 2–7: Schema method generated from worked example

PET can only learn a heuristic for an operator if the purpose of the operator is understood. Initially, PET is restricted to learning operators which achieve a goal state. However, the problem states covered by these heuristics are learned as sub-goals, such that PET can then learn heuristics which achieve these sub-goals. Operator sequences thus grow incrementally.

In order to learn heuristics which recommend operators and operator sequences for problem solving, PET makes use of a facility for learning about operator transformations and propagating this knowledge back through the operator sequence.

Most learning systems hide the operator semantics in code. PET, however, aims to learn the *effects* of an operator and represent this in terms of a *relational model.* The relational model describes links between parts of the problem description before and after the operator application, in order to refer to the *transformation* produced by the operator. The relational model is then used as part of a back propagation process for generalising the application of what are called *episodes*, *ie* sequences of operators for solving a particular task.

Consider the construction of a relational model for the following operator [17]:

OP: $\int x^n dx \rightarrow \frac{x^{n+1}}{n+1} + C$

Assume "$+C$" is dropped for simplicity.

The problem states before and after the application of operator, OP, are represented by parse trees shown in figure 2–8. The relational model comprises links between leaves of both parse trees. The links are provided by specific relations taken from domain dependent background knowledge, in this case, relations about the domain of mathematics. PET uses the relations $eq(X,Y)$, $suc(N,M)$, $sum(L,M,N)$, $product(L,M,N)$ and $derivative(X,Y)$. The links are determined by searching for successful occurrences of the above relations between the two problem states. Figure 2–8 shows the resulting relational model.

---

[17] This example is taken from [Porter 84].

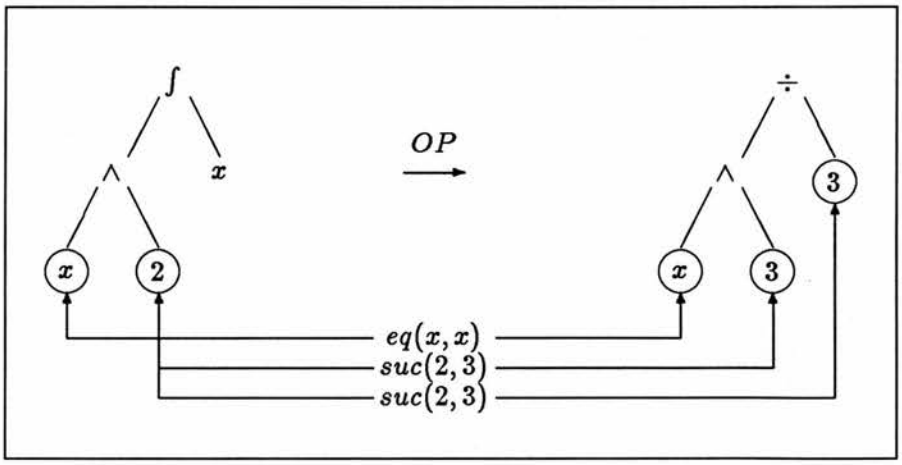


Figure 2–8: A relational model for OP

PET makes use of the relational models in order to learn heuristics for operator sequences. This process is now discussed in the context of another example from [Porter 84] [18]. The example comprises two operators, $OP1$ and $OP2$ in the episode $[OP2, OP1]$, applied to the integration problem, $\int \sin^6 x . \sin x \; dx$. Assume that from prior training for the operator

$OP1$: $sin^2 x \rightarrow 1 - \cos^2 x$

PET has acquired the following relation model, shown in figure 2–9.

Now PET is given advice by the user to apply the operator

$OP2$: $\sin^n x \rightarrow (\sin^2 x)^{\frac{n}{2}}$

to the integration problem, yielding $\int (sin^2 x)^3 sinx \; dx$.

Since PET can only acquire a relational model for this rule if it achieves a known sub-goal, then it must find a way of connecting the preconditions of $OP1$, denoted by the left parse tree of figure 2–9 with the postconditions for $OP2$. The initial version of relational model for the operator $OP2$ is shown in figure 2–10.

---

[18] This example originally came from [Utgoff 84].

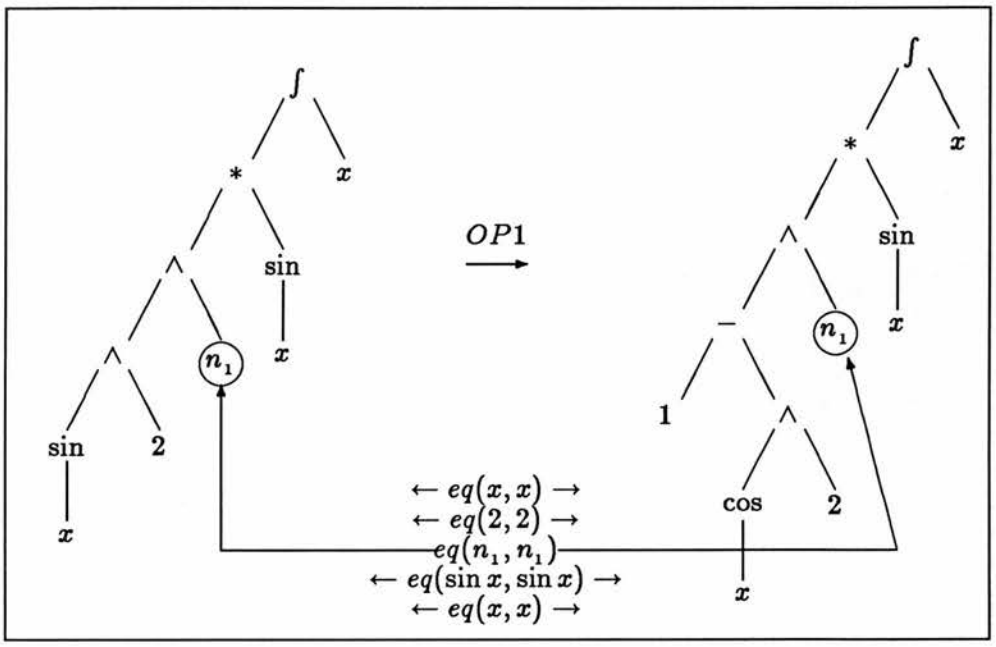


Figure 2–9: Relational model for $OP1$

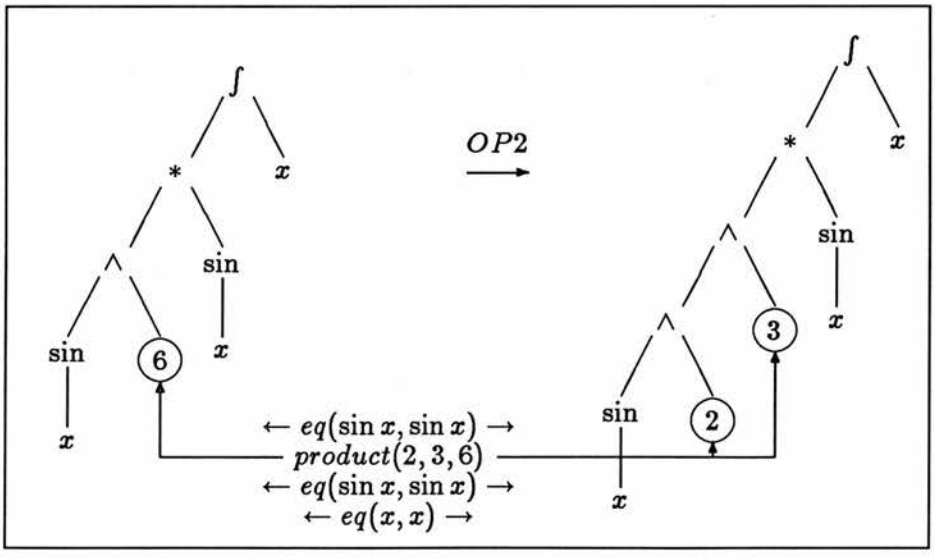


Figure 2–10: Relational models for $OP2$

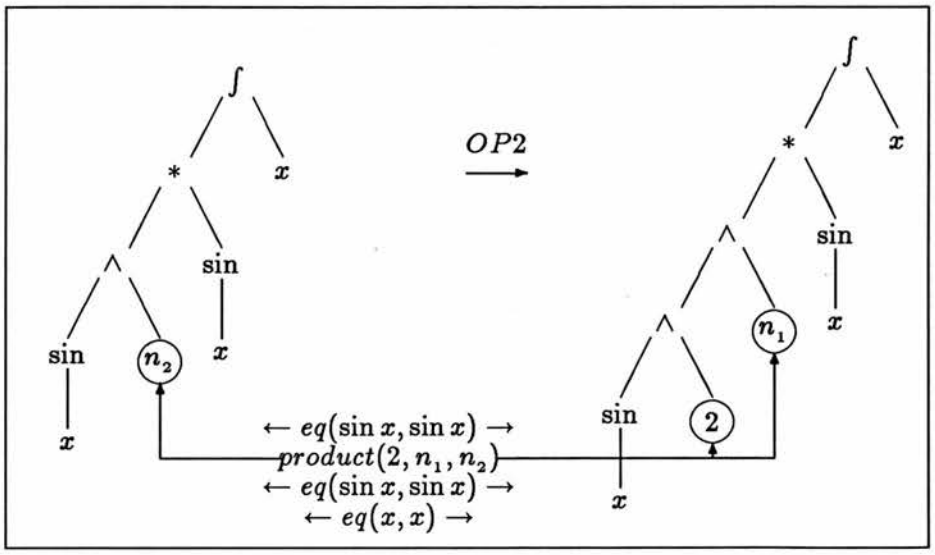


Figure 2–11: Final relational model for $OP2$

PET makes use of a back propagation and perturbation process to generalise the application of the episode $[OP2, OP1]$. The back propagation process involves matching the postconditions of the operator $OP2$, denoted by the right parse tree, with the preconditions of the operator $OP1$, denoted by the left parse tree. This results in binding the variable n_1 with 3, suggesting that $OP2$ is over-specific. The term 3 belongs to the relation, $product(2, 3, 6)$, of $OP2$. The other values of this relation are perturbed and checked for validity. The perturbation of these values involves choosing numbers similar to these values, *eg* for 3 choose 1,2,4,5, and then check whether the relation still holds. The relation is generalised to $product(2, n_1, n_2)$, which corresponds to the concept $even\_integer(n_2)$. Figure 2–11 shows the resulting final relational model for $OP2$.

PET, like LEX2, learns about the applicability of plans and their operators by a combination of a back propagation and perturbation process. PET also learns about the effects of operators by determining the relationship between the states before and after the application of each operator. This provides information about the contribution of each operator within a sequence. However, it makes no explicit reference to the internal structure of the plan and so, no knowledge

about the structure is acquired, except the ordering of the operators in the episode, or sequence.

2.3.5 EBG

For a while in the early 1980s the constraint back propagation technique was adopted by many members of the EBL community for learning control knowledge [Mitchell 83b,Minton 84,Porter 84,Puget 87]. It had some success for a game playing domain, Go-Moku [Minton 84,Puget 87], as well as for problem solving, *ie* symbolic integration [Mitchell 83b,Porter 84]. However, by adopting this technique alone, undue emphasis was placed on learning only one form of control knowledge, *ie* the weakest preconditions for a sequence of operators.

Having been pre-eminent in establishing EBL as a major area within machine learning, the group at Rutgers University, under Mitchell, decided to consolidate the various ideas in EBL within a unified program, called EBG. EBG reflected their ideas on what they have now termed *Explanation-Based Generalization* and has been applied to many classic learning examples. EBG brings together the work on constraint back-propagation [Mitchell 83b] with other work on re-expressing concepts, based on the *operationality criterion* [Keller 80,Mostow 81]. [19].

The operationality criterion defines terms in which the concept must be expressed, such that it becomes *operational*, *ie* examples of the concept are efficiently recognisable.

---

[19]This work acted as a catalyst for other work which either competed with or complemented the EBG technique [Mooney 86,DeJong 86,KedarCabelli 87,Prieditis 87]. Comparison were made with global substitution in EGGS [Mooney 86,DeJong 86]; resolution theorem proving in PROLOG-EBG [KedarCabelli 87] and a combination of EBG with partial evaluation [Prieditis 87]. Claims have been made by the various authors about the greater efficiency and expressibilty of these other approaches compared with EBG.

Given:

- *Goal Concept:* A concept definition describing the concept to be learned. (It is assumed that this concept definition fails to satisfy the operationality criterion.)
- *Training Example:* An example of the goal concept.
- *Domain Theory:* A set of rules and facts to be used in explaining how the training example is an example of the goal concept.
- *Operationality Criterion:* A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed.

Determine:

- A generalisation of the training example that is a sufficient concept definition for the goal concept and that satisfies the operationality criterion.

Table 2–1: Explanation-Based Generalisation Problem

The EBG program addresses the explanation-based generalisation problem in the manner described in table 2–1 [20].

EBG incorporates the back propagation technique from LEX2 and it benefits and suffers from the same advantages and disadvantages. However, it does introduce the operationality criterion which aims to introduce the notion of *utility* of the learnt plan or operator. This idea of utility is developed further by other learning programs which are discussed next.

---

[20]This description of the explanation-based generalisation problem is taken from [Mitchell 86].

2.3.6 PRODIGY

The PRODIGY system has been developed by a large team of researchers at CMU [Minton 87]. It comprises various learning and problem solving systems, including an EBL facility for acquiring *effective* control rules. The emphasis is on learning control knowledge that hopefully improves problem solving performance. PRODIGY addresses what has been termed the *utility* problem. A learning system should not just add more control knowledge indiscriminately. It should take into account the costs of adding the knowledge and measure its benefit to the overall problem solving system.

PRODIGY has an explicit utility measure for evaluating control rules. The utility is determined by the cumulative cost of matching the rule, versus the cumulative savings in search time it produces. The cost/benefit formula is given below

$$Utility = (AvrSavings \times ApplicFreq) - AvrMatchCost$$

where

$AvrMatchCost$ = the average cost of matching the rule,

$AvrSavings$ = the average savings when the rule is applicable,

$ApplicFreq$ = the fraction of times that the rule is applicable when tested

When a rule is learnt from an example, the costs and benefits for that rule can be estimated and validated during subsequent problem solving. Only learnt rules with high utility are kept. Thus, PRODIGY's utility analysis provides a means of improving problem solving performance by removing poor rules.

2.3.7 MetaLEX

The MetaLEX learning system developed by Keller at Rutgers is derived from the original work on LEX and LEX2 [Keller 87]. Some of the ideas in MetaLEX were also incorporated in the EBG program. MetaLEX addresses a type of knowledge

transformation learning task called a *concept operationalization* task [Keller 80]. The task involves re-expressing a given, but unusable, concept description into a form that improves the performance of the problem solving system, SOLVER.

The notion of an operationality criterion is extended to incorporate not only a description of *operational* terms, but also performance objectives that must be met, as a result of the addition of the newly learnt concepts. Such performance objectives are based on values of two measures derived from applying the learnt concept to a set of existing training examples

- Efficiency Level **t**: SOLVER is *efficient* if the cumulative cpu time spent solving the problems in the training set is less than t.

- Effectiveness Percent **p**: SOLVER is *effective* if it successfully solves at least $p\%$ of the problems in the training set.

Provided the performance objectives are achieved, the learnt concept is considered *useful* and the performance of SOLVER is said to have *improved.* The performance objectives also provide a means of stopping the learning process, a facility which has been desired by many machine learning researchers.

Keller's work on MetaLEX makes use of evaluation criteria to improve the performance of its problem solving component. However, whereas PRODIGY performs its utility analysis after the rule has been learnt, MetaLEX's evaluation criteria are actually part of the learning process. The learnt rule is not considered *operational* until it meets various performance objectives which are part of an enhanced notion of the operationality criteria, as in EBG.

Program	1	2	3	4
MACROPS	○	●	●	
LEX2	●			
LP	●	●	○	
PET	●	○		
EBG	●			○
PRODIGY	●			●
MetaLEX	●			●

Table 2–2: Comparison of learning programs

2.4 Conclusions

The comparison of the learning programs (see table 2–2) shows which type(s) of control knowledge is learnt by each program. The various sorts of control knowledge are numbered as follows

1. applicability of plans and operators
2. contributions of operators within plans
3. explicit structure of plan
4. cost/benefit analysis

The solid discs denote that the program is able to learn that type of control knowledge, as has been described in the chapter. The circles denote that only some part of the control knowledge is learnt by the program.

Both MACROPS and LP come out well in the comparison. They are able to learn two types of control knowledge well and make an attempt at learning one of the others.

MACROPS does well because of its triangle table for describing the internal structure of the plan and the contribution of its operators. However, its representation of the applicability of the plan at the various stages is certainly not as expressive as can be obtained with the constraint back propagation technique.

LP does well because the meta-language provides a very expressive means of describing not only the applicability of the operators but also their contribution. Although, LP does not currently express enough strategic knowledge and so does not adequately represent the internal structure of the plan, it promises the best starting point for further development, for the following reasons

- the control knowledge learnt is much more general than for any of the other learning programs,

- it can learn control knowledge from solutions where not all the operators are known,

- extending precondition analysis to deal with some of the above limitations does not seem too difficult a task.

All the other programs involve some form of constraint back propagation. Therefore, they are all able to represent the applicability of operators and plans very well. However, they do not really tackle the representation and learning of other types of control knowledge, except for a cost/benefit analysis.

This last type of control knowledge has only recently been adopted within the EBL community. The discussion about utility analysis within the area of machine learning and the description of the two learning programs, PRODIGY and MetaLEX, has only been touched upon within this chapter. It is not the intention to pursue the representation and learning of this type of knowledge in the rest of the thesis. The use of utility analysis within machine learning is still very limited and an adequate analysis can not be made yet. It is briefly characterised here for completeness.

Summary of Chapter

In this chapter the following has been presented and discussed

- A discussion of the representation of control knowledge is presented. The distinction made between the use of object and meta-level languages for describing control knowledge. Various types of control knowledge are presented
  - applicability of plans and operators
  - contributions of operators within plans
  - explicit structure of plan
  - cost/benefit analysis
- Several learning program are discussed with particular emphasis on what type of control knowledge they learn and how. These are presented in a chronological order, to reflect the development of the EBL approach.
- A comparison is made of these learning programs, with reference to the types of control knowledge that are learnt.

Chapter 3

Learning Proof Plans: A Domain for EBL

3.1 Introduction

This chapter sets the scene for the latter part of the thesis, in particular by describing the domain to which the new EBL technique, described within this thesis, is applied and tested. A domain is presented for which a great deal of knowledge is already known and, yet, which has not been dealt with much by the machine learning community. The domain of interest is that of theorem proving, in particular, the learning of strategies for proving theorems or *proof plans.*

The notion of proof plans and the idea of using such plans for proving new theorems is not commonplace, even within the theorem proving community. So before I discuss issues relating to the learning of such plans, it important for me to answer a few questions relating to proof plans: What are proof plans? How do they help speed up the theorem proving process? Why do they provide an excellent medium for trying out the extended EBL approach? These issues are discussed in section 3.2.

Section 3.3 expands the discussion further by providing a survey of past efforts in representing proof plans and motivates the approach adopted within this thesis for representing such plans by discussing meta-language issues.

Section 3.4 provides more concrete examples of proof plans and the sorts of proofs from which they can be learned. The NuPRL proof development system is introduced. This provides the logic, proof operators and the theory within which the proofs are developed, from which proof plans are learned. A meta-language for describing the required control knowledge is proposed and two example proofs are described using this meta-language. Two resulting proof plans are described.

3.2 A Domain for Learning Control Knowledge

3.2.1 Motivation for proof plans

Proving theorems can often be a non-trivial process even for experienced mathematicians and logicians. Proofs of complex theorems can run into pages of proof transformations, rewritings and applications of lemmas and definitions. Some proofs of theorems have similarities, *ie* parts of proofs may be repeated in other proofs or even within the same proof. For an automatic theorem prover to reprove many of the steps again and again, either within the same proof or for another theorem, seems wasteful.

The simplest approach to reduce such wasted effort is to store proofs of every theorem and apply a proof from the collection of proofs, whenever it is required for another proof. The idea being that if the collection is large enough there is bound to be a proof that will prove the current theorem or that would be useful during the proof [1]. This option is a non-starter, for efficiency reasons in both time and space. Storing a collection of proofs of every theorem would certainly consume an excessive amount of space. If the collection of proofs was very large,

---

[1]Lemmas are often stored and used for trivially proving parts of proofs that have already been proved before. This is particularly beneficial for much larger proofs. However, there is no mechanism for deciding which lemmas to store and which to throw away.

then choosing the correct proof to apply might require a lot of search, especially if the collection contains proofs of trivial theorems. When attempting a proof of another reasonably trivial theorem, the search process may actually take much more time than proving it from scratch.

The key is to capture the strategies involved in achieving the proofs of these theorems. The term *strategy* refers to the set of proof operators required to proof a theorem, together with control knowledge which describes the applicability of these operators and their contribution to the proof as a whole. This definition is expanded further in this chapter.

Many of the proofs have the same strategy because properties of the theorem are the same. By storing these strategies together with their specifications, *ie* a description of the properties of the theorems to which they have been applied, these *proof plans* may be applied further to other theorems which match the specifications. The theorem proving process now involves checking the set of proof plans for a strategy which applies to the current theorem.

The proof plans stored in this way are by no means perfect. They cannot be guaranteed to succeed even when their specifications match the current theorem. Proof plans record the success of previous proof attempts, but can often be over-general in their applicability.

3.2.2 The structure of a proof plan

A proof plan should comprise a *set* of proof operators and a strategy for applying them. Based on the analysis in previous chapters in this thesis, a strategy for applying such proof plans could include the following control knowledge:

- when to apply the proof operators or the entire proof plan,
- what each proof operator contributes to the proof plan,
- the structure of the proof operators within the proof plan

Knowledge about when to apply the operators is provided by the preconditions for each proof operator. The preconditions describe properties of the state of the proof that the particular operator applies to. Preconditions could be described for each level of the proof plan, such that the applicability for each stage of the plan is represented.

The contribution of each proof operator is provided by postconditions or effects of the proof operators. The postconditions describes properties of the state of the proof that the operator produces if it applies successfully. The effects describes the state transition that occur, as a result of applying the proof operator. The contribution of each proof operator provides useful control knowledge for patching the proof plan, if it cannot be applied directly.

The structure of the proof operators provides important information about the order in which the operators should be applied. The structure depends entirely on how the theorem is proved. For a trivial theorem, its proof may be represented by a linear sequence of proof operators. For most theorems, the proof is represented by a more complex tree of proof operators. For the purposes of this discussion and the rest of the thesis, the structure is restricted to a either sequence or a tree of proof operators.

3.2.3 A good domain for EBL

Research in the theorem proving community has resulted in a large set of proof operators available in the literature. Representing the specifications of such operators is a matter of current research within the Mathematical Reasoning Group (MRG) at Edinburgh.

Although, there very few collections of proof plans, their use for theorem proving promises a better understanding of the theorem proving process and, eventually, a significant improvement in performance for proving theorems. The learning of proof plans from examples proofs, using an EBL techniqe, provides the

necessary means for explaining the strategy involved in the proof and generalising the plan, as much as possible [2].

Thus, EBL is the best approach for learning such plans, because it is able to use the existing domain knowledge in the form of known operators and a language for describing the specification of these operators.

The domain of theorem proving provides a rich domain for plans and strategies. The structure of such plans, as we shall see later on with an example, are much more complex than the simple sequences of operators used in the LP project.

3.3 Previous Research

The notion of the proof plan is not very common even in the theorem proving community. This may seem surprising since all theorem provers require strategies to prove even the most trivial theorems. However, following a strategy and describing a strategy are two very distinct tasks. Most theorem provers follow strategies which have been programmed into them by their creators and which are not mentioned or reasoned about in any explicit manner. Some of these theorem provers have been considered very successful. The main benefit of such work has been the characterisation of large sets of proof operators. However, it is very difficult to find work in theorem proving research that describes any of the strategies which it adopts when proving a theorem.

Some exceptions are now described.

3.3.1 LCF

The LCF work of the 1970s does incorporate the notion of proof tactics and provides a metalanguage, ML, for describing the application of proof steps. Proof

---

[2] This idea was first promoted in [Desimone 89].

tactics are expressed in ML, which is a fully higher-order functional programming language.

Complex proof tactics can be built up from simpler ones using higher-order functions, called *tacticals*. For example, (INDUCTION ORELSE CASES) THEN SIMPLIFY [3] is an ML expression describing a strategy which first tries INDUCTION (a given strategy), if that fails CASES is tried and then the resulting sub-goals are passed to SIMPLIFY. The infixed binary operators, ORELSE and THEN are tacticals. These are used to *glue* together existing proof plans to form more complex proof plans.

However, the proof tactics make no reference to the contribution of each of its operators. The structure of the proof tactic is held together with the tacticals, but this produces a rather rigid structure. Thus, the proof tactics are not flexible enough to be built by plan formation or to be patched if they do not apply directly to other relevant proofs. The NuPRL proof development system also provides a representation of proof tactics. This is described later on in this chapter.

3.3.2 IMPRESS

Previous research by Bundy and Sterling resulted in IMPRESS, a program which verified simple logic programs [Bundy 88c] Following the ideas of Boyer and Moore [Boyer 79], Darlington and Burstall [Darlington 81], IMPRESS extended to logic programs the use of the recursive structure of programs to guide the process of induction. In particular, it investigated how different parts of the recursive definition might contribute to the proof.

The IMPRESS work in the early 1980s provided a language for describing parts of a proof that were relevant and at a high enough level for describing a proof plan. However, it did not go far enough in looking at the applicability and

[3]This example is taken from [Bundy 84].

contribution of the proof operators nor the structure of the proof plan. Such control knowledge is considered essential for describing the strategies involved in proofs. As a result, the IMPRESS work did not address all the issues necessary for constructing proof plans.

Nevertheless, this work has identified a proof plan for proving the validity of formulae involving simple implications. This has been called the IMPRESS proof plan. A proof from which this proof plan can be extracted is described later on in this chapter in section 3.5.3.

3.3.3 MT

MT was developed within the Edinburgh Mathematical Reasoning Group in the early 1980s [Wallen 83b,Wallen 83a]. In MT, proof plans, constructed on the basis of properties of the theorems to be proved, provide the system guidance at two levels. At the global level, the proof plans indicate how the main proof may be decomposed into several component proofs, *eg* the base and step cases for inductive proofs, and each carried out by a separate proof module.

At the local level, the proof plan introduces constraints on the form of the proof tree, generated in each module. This reduces the complexity of the proofs required and allows for the application of special-purpose methods to isolate areas of the main proof. In particular, various forms of the resolution inference rule are used to perform refutation proofs.

MT makes use of meta-level axioms, associated with each proof module, to specify forms of the proof that could be generated within the module. The meta-language in MT is restricted to Horn clauses and a PROLOG-like interpreter.

However, although MT promised to provide a good representation for proof plans at a global level, the research eventually digressed into much lower-level issues [Wallen 86,Wallen 87b,Wallen 87a], and so did not adequately address the more higher-level control issues.

3.4 Constructing Proof Plans

So far the motivation for proof plans has been discussed together with a description of some previous attempts at capturing some of the components of a proof plan. In most cases, they stressed the *representation* of some aspect of proof plans. However, none of them tackled the central problem of representing adequate control knowledge, especially from the perspective of making use of it for learning and executing proof plans.

Learning and executing proof plans requires a language that provides enough flexibility to cope with many roles. Such a language must be capable of describing

- properties of the state that are relevant for the application of each stage of the plan;
- the contribution of each operator at each stage of the plan
- the internal structure of operators within the proof plan

In the rest of this chapter, the proof environment, within which the proofs are developed, is described together with a simple meta-language for describing the relevant properties of the proof steps. Two example proofs are then presented [4] and two proof plans are extracted from these proofs. These examples provide the basis for testing the new EBL technique described in chapters 5, 6 and 7.

3.4.1 NuPRL Environment

The NuPRL interactive proof development system was originally designed at Cornell University [Constable 86] and further enhanced at Edinburgh within the

[4] These have been developed using the NuPRL proof development environment and with the restrictions of the simple meta-language in mind.

MRG [5]. NuPRL provides useful tools for developing proofs of theorems. These tools include:

- a set of proof operators,
- various editors for inputting the theorems to be proved and any new proof operators,
- a mechanism for checking the correct application of the proof operators

Theorems are written in terms of the Martin-Löf intuitionistic type theory [MartinLof 79]. The details of this theory are not needed for the analysis in this chapter. However, sufficient understanding of the terminology is required in order to follow the discussion, later on in the chapter, about a proof plan that can be extracted from the example proof.

The constructive nature of the theory allows functions to be extracted from proofs involving existentially quantified terms, *ie* existence proofs. As a result of generating an existence proof, not only is a theorem proved valid within the theory, but also functions may be extracted from the proof steps. Thus, if one considers the theorem to be proved, as a specification for some function, say, of the form,

$$\forall Inputs\ \exists Output.Spec(Inputs, Output)$$

then the algorithm, *alg*, may be extracted from the proof [6], such that

$$\forall Inputs.Spec(Inputs, alg(Inputs))$$

---

[5]This has resulted in two new programs: OYSTER, a reconstruction of the original NuPRL proof development system, and CLAM, a testbed for experimenting with proof plans that makes use of OYSTER [Bundy 88d,Bundy 88e].

[6]The NuPRL interactive proof development system is being used [Bundy 86] as a tool for tackling the two tasks of program synthesis and verification, at the same time.

The proof procedure in NuPRL involves the application of proof operators which refine the theorem to be proved into a set of sub-goals to be proved. This refinement procedure is similar to a goal reduction process, with the exception that at each refinement step there are a set of hypotheses upon which the refinement has been based. Thus, for each goal or sub-goal to be proved, there may also be a set of hypotheses that may be used to aid the choice of subsequent proof operators. Indeed, the hypotheses play an important role in specifying the application of the proof operators. Termination of the proof is obtained when the resulting formula eventually matches a lemma, or matches one of the hypotheses. The result of the proof is a tree which displays the refinement steps together with the relevant proof operators.

The proof operators involved in the enhanced NuPRL comprise both refinement rules and proof tactics. The refinement rules involve simple transformations of the goal into a set of sub-goals to be proved and declare the hypotheses upon which the refinement has been based. Proof tactics represent a more complex transformation of the goal involving some combination of known refinement rules and other proof tactics. The result of a proof tactic is a set of sub-goals which are to be refined further, but should bring the proof closer to termination, *ie* match a lemma or one of the hypotheses.

These proof operators are specified by preconditions and effects which are described in meta-level terms. The next section describes the proposed simple meta-theory. The meta-theory includes a language of meta-level terms, which describe the specifications of the operators and the operators themselves, which are involved in the examples shown later in this chapter.

3.4.2 Proposed Meta Theory

The meta-theory described here is constrained to deal only with the examples presented in the thesis. However, it describes proof plans well enough to test the new learning program, described in the following chapters, on the examples presented here. Developing an complete meta-theory that can deal with many

complex proofs is not the subject of this thesis. Such research work is currently being undertaken within the MRG.

The meta-theory, adopted within this thesis, comprises a language of meta-level terms for describing the specifications of proof operators and a set of the proof operators themselves.

The meta-level terms represent the preconditions and effects of the proof operators [7]. They are of the form, `predicate(Arguments)` [8].

Preconditions

The preconditions represent properties of the state of the proof. The predicates of the following meta-level terms, describing such preconditions, represent these properties. The arguments of the meta-level terms denote the object-level states or partial states to which the properties refer.

`goal(Formula)` – states that `Formula` denotes the formula to be proved.

`hypothesis(Hypo,Hyp_list)` – states that the hypothesis denoted by `Hypo`, is a member of the current hypothesis list, denoted by `Hyp_list`.

`decompose(Sentence,Prefix,Matrix)` – states that `Sentence` can be decomposed into a list of quantifiers, `Prefix`, and another formula, `Matrix`.

`contain(Formula,List_of_vars)` – states that contained within `Formula` there is a list of variables represented by the list, `List_of_vars`.

`exp_at(Formula,Position,Expression)` – expands the `contain` condition by describing the `Position` of the `Expression` within `Formula`.

---

[7] The meta-language presented here is a simplified version of a more complete meta-language developed by Bundy [Bundy 88a].

[8] The PROLOG convention is adopted of predicates written in lower-case letters and the arguments which represent variables, starting with upper-case letters.

`universal(List_of_vars,Formula)` – states that `Formula` contains a list of universally quantified variables, `List_of_vars`.

`existential(List_of_vars,Formula)` – states that `Formula` contains a list of existentially quantified variables, `List_of_vars`.

`prim_rec(Formula,Argument)` – states that the `Formula` is primitively recursive in the argument position `Argument`.

`implication(Formula,Antecendents,Consequents,Preconditions)` – states that the `Formula` comprises an implication between the `Antecedents` and `Preconditions` and the `Consequents`.

Effects

The effects represent the state changes after the application of an operator. The predicates of the following meta-level terms, describing such effects, represent these state changes. The arguments of the meta-level terms denote the object-level states or partial states over which the state changes occur.

`replace_all(Variable,Value,Old_formula,New_formula)` – states that all occurrences of `Variable` in `Old_formula` are replaced by `Value`, resulting in the new formula `New_formula`.

`include(List_of_hyp,Old_hyplist,New_hyplist)` [9] – states that the list of hypotheses, `List_of_hyp`, is appended to the old hypothesis list, `Old_hyplist`, which produces the new hypothesis list, `New_hyplist`.

---

[9] For the purposes of the program, this effect is a shorthand for many `include` effects, where the number depends on the how many new hypotheses are added to the old hypothesis list.

`remove_quantifiers(Sentence,Quantifiers,Formula)` – states that the following quantified terms, `Quantifiers`, have been removed from `Sentence`, resulting in the new formula `Formula`.

`rewrite(Position,Term,Formula1,Formula2)` – states that the expression in `Formula1` located in position `Position` has been rewritten with `Term`, resulting in the new formula `Formula2`.

Proof Operators

This section describes a set of proof operators, together with their preconditions and effects. Note that some proof operators have more than one set of effects, relating to the sub-goals that remain to be proven. Therefore, each set of effects refers to the state transitions that occur between the initial goal to be proven and the relevant sub-goals, which result after the application of the proof operator.

The following proof operators are required for the example proofs, described in this chapter:

`parm_tac` – an operator for stripping away the universal quantifiers from the front of the formula,

preconditions
`goal(Fm)`
`decompose(Fm,QTerm,Form)`
`universal(Set_of_Vars,Fm)`
`contain(Form,Set_of_Vars)`
effects
`include(Set_of_Vars,Hyplist1,Hyplist2)`
`remove_quantifiers(Fm,QTerm,Form)`

`induct` – an operator for generating the base and step case for performing a proof by induction

preconditions

```
goal(Fm)
hypothesis(Var,Hyplist1)
universal([Var],Fm)
contain(Fm,[Var])
```

effects 1

```
replace_all(Var,nil,Fm,BFm)
```

effects 2

```
replace_all(Var,[H|T],Fm,SFm)
include([H,T,IFm],Hyplist1,Hyplist2)
replace_all(Var,T,Fm,IFm)
```

`instantiate_existential_goal` – an operator for stripping away existential variables from the front of the formula in the goal and instantiating the variable in the formula to a value,

preconditions

```
goal(Fm)
decompose(Fm,QTerm,Form)
existential([Var],Fm)
contain(Form,[Var])
hypothesis(H,Hyplist)
hypothesis(T,Hyplist)
```

effects

```
replace_all(Var,[H|T],Form,Form2)
```

`instantiate_existential_hypothesis` – an operator for stripping away existential variables from the front of a formula in the hypothesis set and instantiating the variable in the formula to a value,

preconditions

```
hypothesis(Fm,Hyplist1)
decompose(Fm,QTerm,Form)
```

existential([Var],Fm)

contain(Form,[Var])

effects

include([Form2,Const],Hyplist1,Hyplist2)

replace_all(Var,Const,Form,Form2)

take_out – an operator for identifying the primitive recursive argument position for the base case of an inductive proof and rewriting this with a term which hopefully terminates the base part of the inductive proof.

preconditions

goal(Bfm1)

exp_at(Bfm1,[N|Posn],nil)

exp_at(Bfm1,[0|Posn],Bfm2)

prim_rec(Bfm2,N)

effects

rewrite(Posn,base(Bfm2),Bfm1,Bfm3)

unfold – an operator which identifies the primitive recursive argument position for the step case of an inductive proof and unfolds the expression with the help of an axiom. This unfolding helps to terminate the step part of the proof by permitting the induction hypothesis to be applied.

preconditions

goal(Sfm1)

exp_at(Sfm1,[N|Posn],[H1|T1])

exp_at(Sfm1,[0|Posn],Sfm2)

prim_rec(Sfm2,N)

effects

rewrite(Posn,step(Sfm2),Sfm1,Sfm3)

separation – an operator which makes use of the inductive hypothesis to prove most or all of the step case of the inductive proof. However, this operator is restricted to formulae which involve implications only.

preconditions

`goal(Sfm3)`

`contain(Sfm3,Ants)`

`contain(Sfm3,Cons)`

`contain(Sfm3,Prec)`

`hypothesis(Iifm,Hyplist2)`

`implication(Sfm3,Ants,Cons,Prec)`

effects

`goal(Sfm4)`

`implication(Sfm4,Ants2,Cons2,nil)`

`notcontain(Sfm4,Ants2)`

`notcontain(Sfm4,Cons2)`

3.5 Example proofs and proof plans

Two proofs are now presented and the relevant proof plans required to prove the associated theorems are discussed in the context of the above meta-theory.

3.5.1 Example 1: INSERT proof

The theorem to be proved here describes a function for inserting an element, a, into the list, x, resulting in the new list, z [10].

The proof is typical of many existence proofs that have been generated with the NuPRL environment. This requires the application of the induction proof operator to split the proof into two parts, the base and step cases, which are pursued independently. The induction is performed on a particular variable

[10]Note that for the example proof only, the previous PROLOG convention of predicates written in lower-case letters and the arguments which represent variables starting with upper-case letters is reversed.

in the formula. This is always performed on a universally quantified variable, not an existentially quantified one. Although there may be many universally quantified variables in the formula, generally only one is chosen as the induction variable. The remainder of the proof in the base and step cases reveals more information about the structure of the formula, particularly the relationships between the induction variable and the other universal and existential variables in the formula.

The full proof of the theorem is not shown here, since this would cover several pages. Enough of the proof is given in order to describe a partial proof plan, which includes the various types of control knowledge that Extended-EBL is able to learn.

For the purposes of the discussion here and in the next chapter, the full expressiveness of the constructive type theory is *not* represented. References to the *types* of terms in the theory are not included in the descriptions of the formulae to be proved. Thus, massive branches of the proof tree, which relate to the proofs of well-formed types within the type theory, are also ignored.

The theorem to be proved is represented by the following formula:

$$\forall a \forall x. \exists z. I(a, x, z) \tag{3.1}$$

The insert function is specified by the expression, $I(a, x, z)$. The partial proof tree is shown in figure 3–1. The full proof is given in appendix A.

Considering the refinement proof procedure as a goal reduction process, the initial goal to be proved is the expression (3.1).

The application of the first proof operator, `parm_tac`, removes all the outermost universal quantifiers from the goal. The previously quantified parameters are now represented as hypotheses of the new sub-goal to be proved [11].

---

[11] These hypotheses are like the parameter declarations of some programming languages, eg, PASCAL.

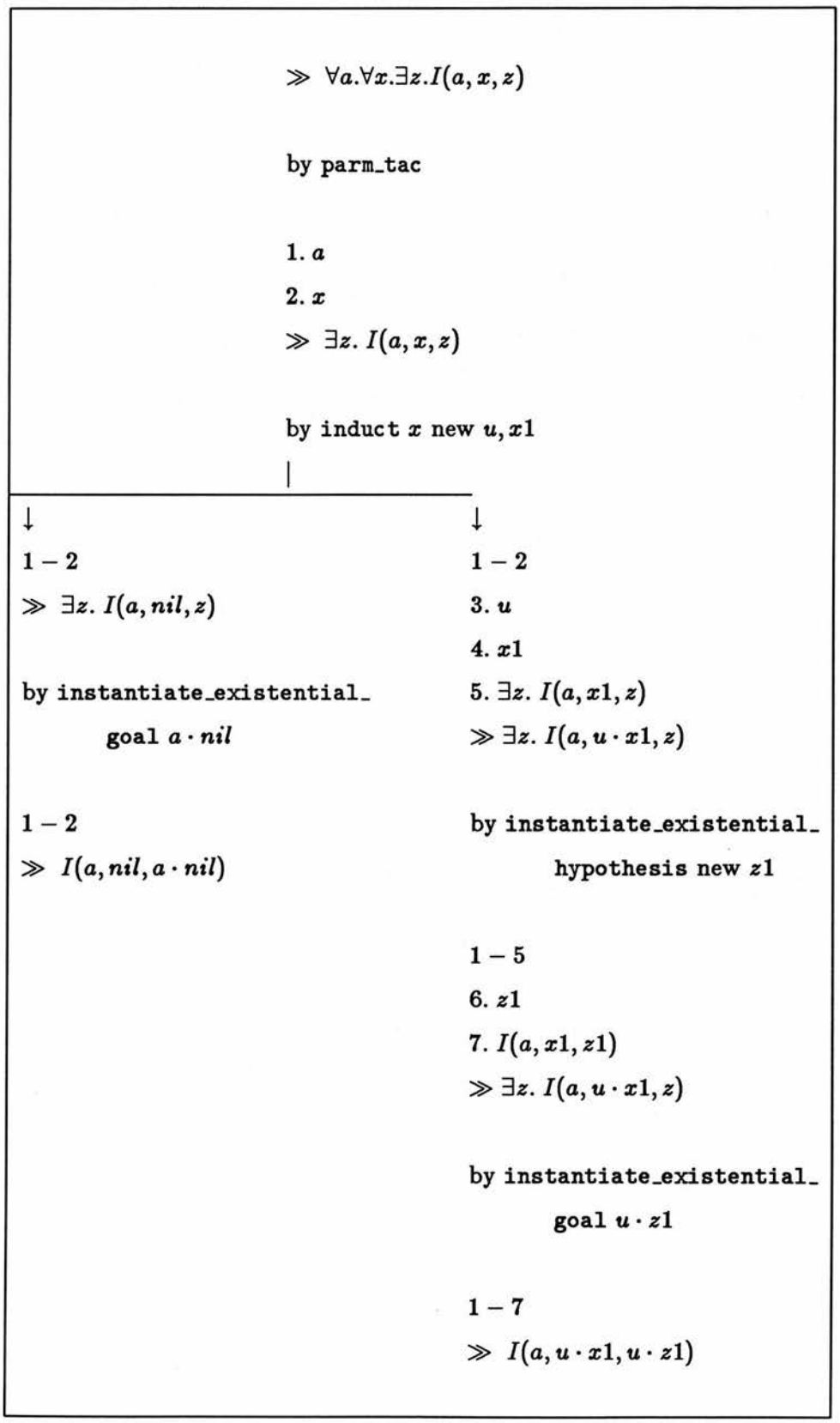


Figure 3–1: Representation of a partial proof tree

The next operator, induct, splits the proof tree into two sub-goals, one of which deals with the base case, represented by the expression, $\exists z.I(a, nil, z)$, and the other the step case, represented by the expression, $\exists z.I(a, u \cdot x1, z)$ [12]. These two sub-goals are produced by replacing the induction variable, represented by the list, x, with the base and step values, nil and $u \cdot x1$ respectively. In addition, for the step case, some hypotheses are added to the hypothesis list. The two hypotheses, u and $x1$ are essential for proving the well-formedness of the expression, $I(a, u \cdot x1, z)$, and so are not discussed here. However, the other hypothesis, $\exists z.I(a, x1, z)$ represents the induction hypothesis, which is required later on in the proof.

For the base case, the next operator involves instantiating the existential variable, z, with the appropriate value, $a \cdot nil$ [13]. This leaves the expression $I(a, nil, a \cdot nil)$ still to be proved. The rest of this branch of the proof tree involves further decomposition of the expression until a termination node is reached. This occurs when the goal to be proved matches one of the hypotheses of the hypothesis set or matches some previously proved lemma.

For the step case, the next two operators perform similiar tasks. The existential variable, z, in both the induction hypothesis, $\exists z.I(a, x1, z)$ and the step goal, $\exists z.I(a, u \cdot x1, z)$ is instantiated by $z1$ and $u \cdot z1$ respectively. This keeps the description of both the new induction hypothesis, $I(a, x1, z)$ and the new step goal, $I(a, u \cdot x1, u \cdot z1)$ at the same level. The rest of the step case proof proceeds until the tree is terminated. Note that this involves the use of the induction hypothesis, $I(a, x1, z)$.

---

[12]The $\cdot$ connective, denotes a *cons* binary operator, such that the list $u \cdot x1$ has u at the head and $x1$ at the tail of the list.

[13]This value is appropriate because the result of inserting a into the empty list, nil, is the list, $a \cdot nil$. Since NuPRL is an interactive proof development system and not an automatic theorem prover, this value has to be provided by the user.

Although the rest of the proof is not shown, there is enough of the proof tree to test the capabilities of Extended-EBL. This is done in the next chapter 6.

3.5.2 INSERT proof plan

The proof plan must describe the structure of the proof operators required to prove the theorem, together with knowledge about: the applicability of each proof operator, the overall plan and the contribution of each proof operator.

In the case of the insert proof, the structure is in the form of a tree of proof operators, where one branch represents the base case and the other the step case, as shown in figure 3–2 and table 3–1. However, note that figure 3–2 only shows part of the required proof plan. The discs represent the preconditions and effects of the operators. The boxes represent the operators themselves. The links labelled with `m` denote the links between the effects of one operator and the preconditions of another.

The proof plan must represent the applicability of this tree of proof operators to formulae involving universally and existentially quantified variables. This can be done with the meta-level preconditions described in section 3.4.2 as follows:

`contain(`$\exists$`z.I(a,x,z),[a,x])`	from $a1$
`universal([a,x],`$\forall a.\forall x.\exists$`z.I(a,x,z))`	from $a2$
`decompose(`$\forall a.\forall x.\exists$`z.I(a,x,z),[`$\forall a.\forall x$`],`$\exists$`z.I(a,x,z))`	from $a3$
`goal(`$\forall a.\forall x.\exists$`z.I(a,x,z))`	from $a4$
`decompose(`$\exists$`z.I(a,x,z),`$\exists$`z,I(a,x,z))`	from $c1'$
`existential(z,`$\exists$`z.I(a,x,z))`	from $c2'$
`contain(I(a,x,z),z)`	from $c3'$

$a1$–$a4$ represent the preconditions of the `parm_tac` proof operator for removing the universally quantifers. $c1'$–$c3'$ represent the other preconditions of the proof plan that identify the existentially quantified variable, z. $b3$ represents a redundant precondition implied by $a1$.

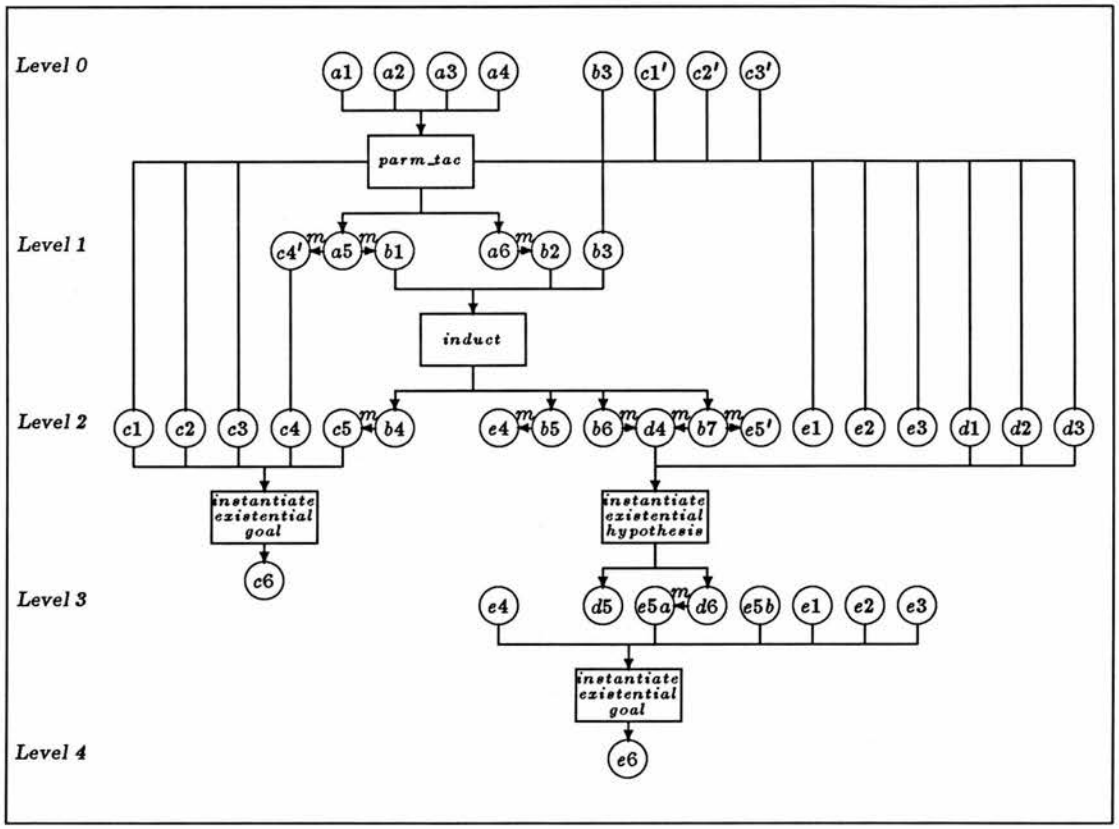


Figure 3–2: Representation of the insert proof plan

Nodes	*Preconditions and Effects*
$a1$	`contain(fm2,[var1,var2])`
$a2$	`universal([var1,var2],fm)`
$a3$	`decompose(fm,qterm,fm2)`
$a4$	`goal(fm)`
$a5$	`include([var1,var2],hyplist1,hyplist2)`
$a6$	`remove_quantifier(fm,qterm,fm2)`
$b1$	`hypothesis(var2,hyplist2)`
$b2$	`goal(fm2)`
$b3$	`contain(fm2,[var2])`
$b4$	`replace_all(var2,nil,fm2,bfm1)`
$b5$	`replace_all(var2,[h1|t1],fm2,sfm1)`
$b6$	`replace_all(var2,t1,fm2,ifm1)`
$b7$	`include([h1,t1,ifm1],hyplist2,hyplist3)`
$c1$	`decompose(bfm1,qterm2,bfm2)`
$c1'$	`decompose(fm2,qterm2,fm3)`
$c2$	`existential([var3],bfm1)`
$c2'$	`existential([var3],fm2)`
$c3$	`contain(bfm2,[var3])`
$c3'$	`contain(fm3,[var3])`
$c4$	`hypothesis(var1,hyplist2)`
	`hypothesis(nil,hyplist2)`
$c4'$	`hypothesis(var1,hyplist2)`
	`hypothesis(var2,hyplist2)`
$c5$	`goal(bfm1)`
$c6$	`replace_all(var3,[var1|nil],bfm2,bfm3)`
$d1$	`decompose(ifm1,qterm2,ifm2)`
$d1'$	`decompose(fm2,qterm2,fm3)`
$d2$	`existential([var3],ifm1)`
$d2'$	`existential([var3],fm2)`
$d3$	`contain(ifm2,[var3])`
$d3'$	`contain(fm3,[var3])`
$d4$	`hypothesis(ifm1,hyplist3)`
$d5$	`replace_all(var3,t3,ifm2,ifm3)`
$d6$	`include([ifm3,t3],hyplist3,hyplist4)`
$e1$	`decompose(sfm1,qterm2,sfm2)`
$e1'$	`decompose(fm2,qterm2,fm3)`
$e2$	`existential([var3],sfm1)`
$e2'$	`existential([var3],fm2)`
$e3$	`contain(sfm2,[var3])`
$e3'$	`contain(fm3,[var3])`
$e4$	`goal(sfm1)`
$e5a$	`hypothesis(t3,hyplist4)`
$e5b$	`hypothesis(h1,hyplist4)`
$e5'$	`hypothesis(h1,hyplist3)`
$e6$	`replace_all(var3,[h1|t3],sfm2,sfm3)`

The proof plan must also identify the contribution of each proof operator. The contribution of a proof operator is described by the link between the effects of one operator and the preconditions of another. For instance, the effects of the induct operator are to replace the induction variable x by *nil* for the base case and $u \cdot x1$ for the step case. These are described by

```
replace_all(x,nil,∃z.I(a,x,z),∃z.I(a,nil,z))          from b4
```

which links to precondition, goal(bfm1), denoted by $c5$, for the base case, and

```
replace_all(x,[u|x1],∃z.I(a,x,z),
            ∃z.I(a,[u|x1],z))                          from b5
```

which links to goal(sfm1),$e4$, for the step case. $b6$ and $b7$ represented by

```
replace_all(x,x1,∃z.I(a,x,z), ∃z.I(a,x1,z))           from b6
include([u,x1,∃z.I(a,x1,z)],[a,x],
            [a,x,u,x1,∃z.I(a,x1,z)])                   from b7
```

define the induction hypothesis, ∃z.I(a,x1,z), and are linked to the preconditions

```
hypotheses([∃z.I(a,x1,z)],[a,x,∃z.I(a,x1,z)])         from d4
hypotheses([u],[a,x,∃z.I(a,x1,z)])                    from e5'
```

respectively.

The complete proof plan identifies the full role of the inductive hypothesis within the proof and the other restrictions on the formula to be proved.

3.5.3 Example 2: ISOLATE proof

The next proof is taken from [Bundy 88c] and involves the proof of correctness of the *isolate* method taken from PRESS. This particular proof has been chosen because it is an example of the sort of theorem that can be proved using the IMPRESS proof plan mentioned earlier. In addition, the proof is short and the application of proof operators is typical of many inductive proofs.

The theorem to be proved is the following:

$$
\begin{aligned}
&single\_occ(X, Lhs = Rhs)\ \&\\
&position(X, Lhs, Posn)\ \&\\
&isolate(Posn, Lhs = Rhs, X = Ans)\\
&\rightarrow solve(Lhs = Rhs, X, X = Ans)
\end{aligned}
$$

where:

- $single\_occ(X, Exp)$ means X occurs only once in Exp;
- $position(X, Exp, Posn)$ means X occurs at $Posn$ in Exp;
- $isolate(Posn, Eqn, Soln)$ means $Soln$ is isolated at $Posn$ in Eqn;
- $solve(Eqn, X, Soln)$ means $Soln$ solves Eqn for X.

The proof is shown in figure 3–3.

The application of the `induct` operator, splits the proof tree into two sub-goals, one which deals with the base case, represented by the expression,

$$
\begin{aligned}
&single\_occ(x, lhs = rhs)\ \&\\
&position(x, lhs, [])\ \&\\
&isolate([], lhs = rhs, x = ans)\\
&\rightarrow solve(lhs = rhs, x, x = ans)
\end{aligned}
$$

and one which deals with the step case, represented by the expression,

$$
\begin{aligned}
&single\_occ(x, lhs = rhs)\ \&\\
&position(x, lhs, [n|posn])\ \&\\
&isolate([n|posn], lhs = rhs, x = ans)\\
&\rightarrow solve(lhs = rhs, x, x = ans)
\end{aligned}
$$

These two sub-goals are produced by replacing the induction variable, $Posn$, with the base and step values [] and $[n|posn]$ respectively. In addition, for the

$\gg$ *single_occ*$(X, Lhs = Rhs)$ &
position$(X, Lhs, Posn)$ &
isolate$(Posn, Lhs = Rhs, X = Ans)$
$\rightarrow$ *solve*$(Lhs = Rhs, X, X = Ans)$

by induct *Posn* new $n, posn$

|

↓

$\gg$ *single_occ*$(x, lhs = rhs)$ &
position$(x, lhs, [])$ &
isolate$([], lhs = rhs, x = ans)$
$\rightarrow$ *solve*$(lhs = rhs, x, x = ans)$

by take_out x 2

$\gg$ *true*

↓

1. n
2. $posn$
3. *single_occ*$(X, Lhs = Rhs)$ &
position$(X, Lhs, Posn)$ &
isolate$(Posn, Lhs = Rhs, X = Ans)$
$\rightarrow$ *solve*$(Lhs = Rhs, X, X = Ans)$
$\gg$ *single_occ*$(x, lhs = rhs)$ &
position$(x, lhs, [n|posn])$ &
isolate$([n|posn], lhs = rhs, x = ans)$
$\rightarrow$ *solve*$(lhs = rhs, x, x = ans)$

by unfold x 2

$1 - 3$
$\gg$ *single_occ*$(x, lhs = rhs)$ &
position$(x, lhs, [n|posn])$ &
isolax$(n, lhs = rhs, lhs' = rhs')$ &
isolate$([n|posn], lhs = rhs, x = ans)$
$\rightarrow$ *equiv*$(lhs = rhs, Eqn)$ &
solve$(Eqn, x, x = ans)$

by separation

$1 - 3$
$\gg$ *isolax*$(n, lhs = rhs, lhs' = rhs')$
$\rightarrow$ *equiv*$(lhs = rhs, lhs' = rhs')$

step case some hypotheses are added to the hypothesis list. The two hypotheses, n and $posn$, and the induction hypothesis,

$$
\begin{aligned}
&single\_occ(X, Lhs = Rhs)\ \&\\
&position(X, Lhs, Posn)\ \&\\
&isolate(Posn, Lhs = Rhs, X = Ans)\\
&\rightarrow solve(Lhs = Rhs, X, X = Ans)
\end{aligned}
$$

which is required later on in the proof.

For the base case, the next step involves two applications of the `take_out` operator to terminate this part of the proof. This occurs because the axiom for *isolate* shows that $isolate([], lhs = rhs, x = ans)$ is trivially true with $lhs = rhs$ equivalent to $x = ans$. Similarly, for $solve(lhs = rhs, x, x = ans)$ with $x \equiv lhs$ and $ans \equiv rhs$. As a result, the other two terms, $single\_occ(x, lhs = rhs)$ and $position(x, lhs, posn)$ are also trivially true.

For the step case, two applications of the `unfold` also make use of the axioms for *isolate* and *solve* to unfold the expressions in the step case, resulting in

$$isolax(n, lhs = rhs, lhs' = rhs')\ \&\ isolate(posn, lhs = rhs, x = ans)$$

from $isolate([n|posn], lhs = rhs, x = ans)$ and

$$equiv(lhs = rhs, Eqn)\ \&\ solve(Eqn, x, x = ans)$$

from $solve(lhs = rhs, x, x = ans)$.

The next operator, `separation`, makes use of the inductive hypothesis to produce two sub-goals which are trivially satisfied by existing axioms. These are

$$
\begin{aligned}
&isolax(n, lhs = rhs, lhs' = rhs')\\
&\rightarrow equiv(lhs = rhs, lhs' = rhs') \text{ and}
\end{aligned}
$$

$$
\begin{aligned}
&single\_occ(x, lhs = rhs)\ \&\\
&position(x, lhs, [n|posn])\ \&\\
&isolax(n, lhs = rhs, lhs' = rhs')
\end{aligned}
$$

$\rightarrow$ *single_occ*$(x, lhs' = rhs')$ &
position$(x, lhs', posn)$

3.5.4 ISOLATE proof plan

For the isolate proof, as for the insert proof, the structure of the proof operators is represented by a tree, as shown in figure 3–4 and table 3–2. This is not surprising, since both involve inductive proofs. However, the isolate proof plan applies only to formulae involving implications of the form:

`Antecedent & Preconditions` $\rightarrow$ `Consequent`

For this particular proof the formula, denoted `Fm1`, is

single_occ$(X, Lhs = Rhs)$ &
position$(X, Lhs, Posn)$ &
isolate$(Posn, Lhs = Rhs, X = Ans)$
$\rightarrow$ *solve*$(Lhs = Rhs, X, X = Ans)$

where

`Antecedent`	denotes	*isolate*$(Posn, Lhs = Rhs, X = Ans)$ & *position*$(X, Lhs, Posn)$
`Preconditions`	denotes	*single_occ*$(X, Lhs = Rhs)$
`Consequent`	denotes	*solve*$(Lhs = Rhs, X, X = Ans)$

`Antecedent` and `Consequent` both contain the induction variable, $Posn$.

The proof plan must reflect all these restrictions on the formula, together with the preconditions of the `induct` proof operator. This can be done with the meta-level preconditions

`goal(Fm1)`	from $a1$
`hypothesis(`$Posn$`,[`$Posn$`])`	from $a2$
`contain(Fm1,[`$Posn$`])`	from $a3$

Nodes	*Preconditions and Effects*
$a1$	goal(fm1)
$a2$	hypothesis(var,hyplist1)
$a3$	contain(fm1,[var])
$a4$	replace_all(var,nil,fm1,bfm1)
$a5$	replace_all(var,[h1|t1],fm1,sfm1)
$a6$	include([h1,t1,ifm],hyplist1,hyplist2)
$a7$	replace_all(var,t1,fm1,ifm1)
$b1$	goal(bfm1)
$b2$	exp_at(bfm1,[n|posn],nil)
$b3$	exp_at(bfm1,[0|posn],bfm2)
$b3'$	exp_at(fm1,[0|posn],fm2)
$b4$	prim_rec(bfm2,n)
$b4'$	prim_rec(fm2,n)
$b5$	rewrite(posn,base(bfm2),bfm1,bfm3)
$c1$	goal(sfm1)
$c2$	exp_at(sfm1,[n|posn],[h1|t1])
$c3$	exp_at(sfm1,[0|posn],sfm2)
$c3'$	exp_at(fm1,[0|posn],fm2)
$c4$	prim_rec(sfm2,n)
$c4'$	prim_rec(fm2,n)
$c5$	rewrite(posn,step(sfm2),sfm1,sfm3)
$d1$	goal(sfm3)
$d2$	contain(sfm3,ants)
$d2'$	contain(fm1,ants)
$d3$	contain(sfm3,cons)
$d3'$	contain(fm1,cons)
$d4$	contain(sfm3,prec)
$d4'$	contain(fm1,prec)
$d5$	hypothesis(ifm,hyplist2)
$d6$	implication(sfm3,ants,cons,prec)
$d6'$	implication(fm1,ants,cons,prec)
$d7$	goal(sfm4)
$d8$	implication(sfm4,ants2,cons2,nil)
$d9$	notcontain(sfm4,ants2)
$d10$	notcontain(sfm4,cons2)

Table 3–2: Preconditions and effects of the proof operators for the isolate proof plan

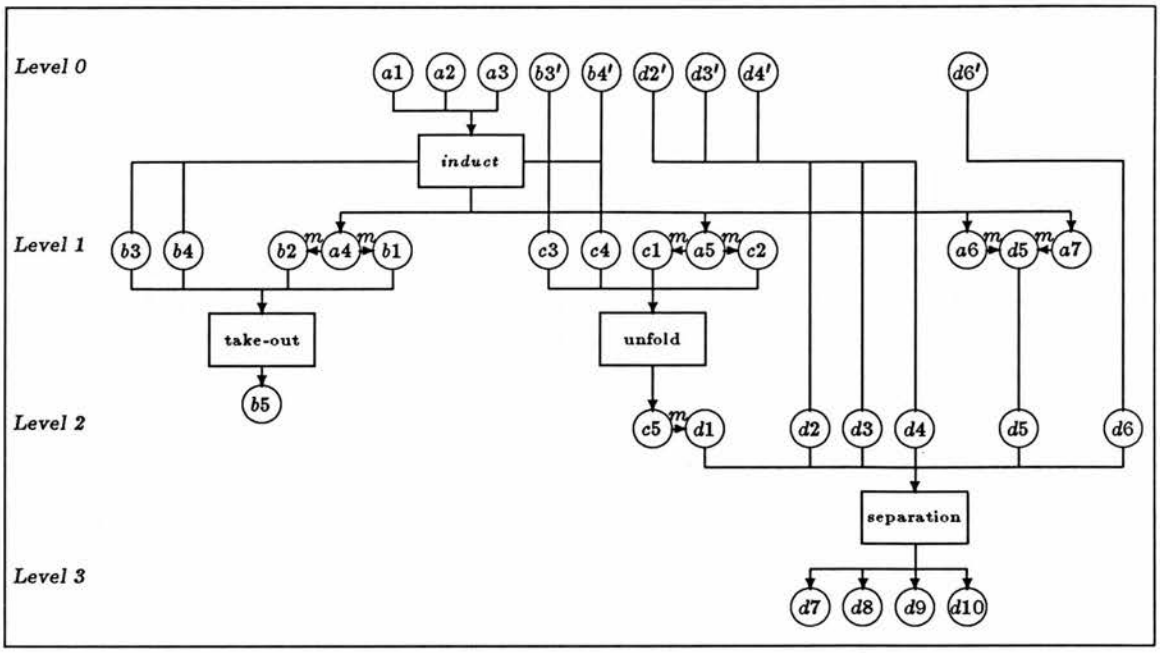


Figure 3–4: Representation of the isolate proof plan

`exp_at(Fm1,Posn1,`$isolate(Posn, Lhs = Rhs, X = Ans)$`)`	from $b3'$
`prim_rec(`$isolate(Posn, Lhs = Rhs, X = Ans)$`,Posn2)`	from $b4'$
`contain(Fm1,Ant)`	from $d2'$
`contain(Fm1,Cons)`	from $d3'$
`contain(Fm1,Pre)`	from $d4'$
`implication(Fm1,Ant,Cons,Pre)`	from $d6'$

where $a1$–$a3$ describe the preconditions of the `induct` operator, $b3'$ and $b4'$ identify the position of the induction variable, $Posn$, and $d2'$–$d6'$ describe that the formula, `Fm1` must involve the simple implication of the form presented above.

The role of the inductive hypothesis is captured by the contribution of the effects

`include([n,posn,IndHyp],[Posn],` `[Posn,n,posn,IndHyp])`	from $a6$
`replace_all(Posn,Posn,Fm1,IndHyp)`	from $a7$

to the precondition, `hypothesis(IndHyp,[Posn,n,posn,IndHyp])`, of the `separation` operator, denoted by $d5$.

Summary of the Chapter

In this chapter the following has been presented and discussed:

- The area of theorem proving is proposed as a domain for learning control knowledge, in particular, for learning strategies for proving theorems or *proof plans.*

- Past efforts are described for representing and using proof plans. The idea of representing knowledge about proof plans and tactics in terms of a meta-level language is discussed and argued for.

- Two necessary tools are required to provide and discuss proof plans

    1. The environment within which the proofs are generated *viz* the NuPRL proof development system.
    2. A meta-language for describing the structure of the proof.

- Two example proofs are described and the proof plan involved in proving the relevant theorems are described in terms of the simple meta-language.

Chapter 4

Reconstruction of Precondition Analysis

4.1 Introduction

The aim of this chapter is two-fold. On the surface, it is a descriptive chapter containing details about a successful EBL technique, Precondition Analysis. However, another important aim of this chapter is to provide a case for considering precondition analysis as a starting point from which to develop and extend the expressibility of the EBL approach, for learning control knowledge. How is this achieved?

Section 4.2 shows how precondition analysis relies on the representation of meta-knowledge. Inference at the meta-level provides, not only a more efficient means of performing problem solving, but also a language for describing more general properties of the problem state. The latter point is extremely important for a machine learning technique.

More details are provided about the precondition analysis technique in section 4.3. Particular reference is made to two pieces of knowledge that are acquired by precondition analysis and how these are used to learn other types of control knowledge.

A reconstruction of precondition analysis is described in section 4.4. This has been implemented in PROLOG and forms the basis for further development, later on in the thesis.

4.2 Role of Meta-Level Inference in PRESS/LP

This section provides a description of object and meta-level knowledge for the domain of algebraic problem solving and shows how meta-level inference plays an important role for learning and problem solving. The research discussed here is based on previous work in the Mathematical Reasoning Group at the Dept. of Artificial Intelligence at Edinburgh [Bundy 81,Sterling 82,Silver 83,Silver 84, Silver 85].

4.2.1 Object and Meta-Level Knowledge

In PRESS and LP, knowledge about the state of the equation and the operations involved in algebraic problem solving are characterised at two distinct levels:

- the *object*-level
- the *meta*-level

as described in chapter 2. The algebraic problems to solve, which are input into PRESS, and the solutions to algebraic problems, which are supplied to LP, are both in the form of algebraic equations. The terms used to describe algebraic equations are called *object*-level terms. The algebraic equation

$$x^2 + 2 \cdot x + 1 = 0$$

contains the object-level terms, $x, 2, 1, 0, +, =, \cdot$. Note that the term 2 is used both as a multiplier and exponent for x.

The set of operations that can be applied to the algebraic equations are also represented at the object-level, in the form of rewrite rules.

The following are examples of rewrite rules

$$U^V \cdot U^W \Rightarrow U^{V \cdot W} \tag{4.1}$$

$$U \cdot N + U \cdot M \Rightarrow U \cdot (N + M) \tag{4.2}$$

$$U^2 + 2 \cdot U \cdot V + V^2 \Rightarrow (U + V)^2 \tag{4.3}$$

$$\log_W U + \log_W V \Rightarrow \log_W U \cdot V \tag{4.4}$$

$$A^U = B \Rightarrow U = \log_A B \tag{4.5}$$

However, the process of guiding the search for solutions and learning about the structure of these solutions is performed at the meta-level [Bundy 81]. Thus, the state of the equation, initially, and at each step in the solution is described in meta-level terms. These terms describe *syntactic* properties about the equation, such as the number of occurrences of the unknown variable in the equation; whether the equation consists of a product or sum of algebraic terms, etc. Such terms provide descriptions of properties of the equation, that are more general than equivalent object-level terms.

Thus, the following equation

$$3 \cdot x + 2 \cdot x + 5 = 0 \tag{4.6}$$

has the following properties, described in meta-level terms

lhs-sum(X,Eqn)	lhs of Eqn contains a sum of terms in X
Multiple-occ(X,Eqn)	multiple occurrences of X in Eqn
rhs-zero(Eqn)	rhs of Eqn is zero

where X refers to the unknown variable, x, and Eqn refers to the algebraic equation 4.6.

Rewrite rules may be composed to form sets of rewrite rules which describe meta-level operation, called *methods*. These methods perform more general algebraic operations than the individual rewrite rules [Bundy 81].

Examples of methods are the following [1]

collection – reducing the number of occurrences of the unknown variable, preferably, but not necessarily, to a single occurrence – the expressions in 4.1, 4.2 and 4.3 belong to the set of *collection* rewrite rules.

attraction – bringing terms containing the unknown variable closer together. This is often necessary before the *collection* method is applied – 4.4 is an example of an *attraction* rewrite rule.

isolation – isolating the unknown variable as the subject of the formula – 4.5 is an example of an *isolate* rewrite rule.

Each method consists of a set of rewrite rules which perform the same meta-level operation on many different object level equations. 4.1, 4.2 and 4.3 show three collection rewrite rules that deal with equations containing multiple occurrences of unknown variables U and V. Another example could involve the isolation method on equations containing sines, cosines and logarithmic functions, each represented by different rewrite rules.

Each method is specified by preconditions and postconditions, described in meta-level terms. These represent properties that should be present before the application of the method, and those that should exist after the method has been applied.

Figures 4–1 and 4–2, which have already been shown in chapter 2, are repeated here to show the meta-level and object-level representation of the solution to the equation

$$\cos(x) + 2 \cdot \cos(2 \cdot x) + \cos(3 \cdot x) = 0 \qquad (4.7)$$

---

[1] A more complete set of methods can be found in [Silver 83,Silver 84,Silver 85]

4.2.2 Inference at the Meta-Level

Without access to meta-level knowledge, generating the solution to algebraic equations involves searching the entire space of rewrite rules for the correct one to apply at each stage of the solution. However, with PRESS, the process of generating a solution is performed entirely at the meta-level.

This process is called *meta-level inference* and involves guiding the search for which method to apply next. The particular method is chosen by matching the meta-level preconditions of the various methods with the description of the current state of the equation. Since a method comprises a set of rewrite rules, then choosing a method constrains the search to the choice of one of the rewrite rules associated with the method. The search for the correct rewrite rule is, thus, reduced to a search at the meta-level for the relevant method. Since the number of methods is much smaller than the number of rewrite rules, then the overall search for the solution is reduced.

The search is constrained further by what is called the *waterfall* strategy [Sterling 82,Silver 85]. The waterfall contains a number of methods. At the top of the waterfall, PRESS checks whether the equation is solved. If not, the equation is passed over the waterfall and a method is selected and applied to the equation. After the application of the method, it is returned to the top of the waterfall and the process is repeated. If none of the methods can be applied, then PRESS backtracks. If the equation still remains in the waterfall then PRESS fails to solve the equation. The selection of methods is based on an enforced ordering, which gives some methods priority over others. The ordering is based on empirical evidence and is wired into PRESS. Methods such as isolation, which lead to a direct solution, are tried first before others.

For the learning in LP, the aim is to learn about the task of algebraic problem solving from single examples of solutions to equations [2]. There are two main learning tasks performed within LP

---

[2]These can be found in most mathematics text books as *worked examples.*

- LP is able to learn information about when to apply these methods and in what order they should be applied to achieve a solution to a particular equation.

- LP is also able to learn new algebraic methods or extend existing methods, whenever it identifies a step which cannot be explained by an existing method.

These two learning tasks are achieved by making use of the meta-level preconditions and postconditions of known methods. This is described in the next section.

Thus, inference at the meta-level helps not only for problem solving, in guiding the search for a solution, but also for providing valuable control knowledge about the specifications of missing methods and about the strategies involved in the example solutions. These ideas are explained further in the next sections.

4.3 Precondition Analysis

The learning process in LP is performed by the *Precondition Analysis* technique [Silver 83,Silver 84,Silver 85], as described in chapter 2. Precondition Analysis contributes to two learning tasks

- *learning schema methods*, by finding the reasons for applying each method at each step of the problem solution and determining a strategy for solving the problem, and

- *learning new methods*, by determining the specifications of unknown methods involved in the solution.

To achieve these two learning tasks, two pieces of information are deduced for each method involved in the solution to the problem

- *Satisfied Preconditions* (S), and

$$\cos(x) + 2 \cdot \cos(2 \cdot x) + \cos(3 \cdot x) = 0$$

(Cosine Rule)

$$2 \cdot \cos(2 \cdot x) \cdot \cos(x) + 2 \cdot \cos(2 \cdot x) = 0$$

(Factorisation Preparation)

$$2 \cdot \cos(2 \cdot x) \cdot (\cos(x) + 1) = 0$$

(Factorisation)

$$\cos(2 \cdot x) = 0 \vee \cos(x) + 1 = 0$$

Solve first factor	*Solve next factor*
$\cos(2 \cdot x) = 0$	$\cos(x) + 1 = 0$
(Isolation)	*(Isolation)*
$x = 90 \cdot n_1 + 45$	$x = 180 \cdot (2 \cdot n_2 + 1)$

Figure 4–1: Worked Example after identification of methods

Name	Satisfied Preconditions	Major Effects
New Method (Cosine Rule)	rhs-zero(Eqn1), lhs-sum(X,Eqn1), multiple-occ(X,Eqn1)	common-subterms(X,Eqn2)
Factorisation Preparation	rhs-zero(Eqn1), lhs-sum(X,Eqn1), multiple-occ(X,Eqn1) common-subterms(X,Eqn1)	lhs-product(X,Eqn2)
Factorisation	rhs-zero(Eqn1), lhs-product(X,Eqn1) multiple-occ(X,Eqn1)	
Isolation	single-occ(X,Eqn1)	(Solution)
Isolation	single-occ(X,Eqn2)	(Solution)
Generating Equation: $\cos(x) + 2 \cdot \cos(2 \cdot x) + \cos(3 \cdot x) = 0$ Unknown: x		

Figure 4–2: Schema method generated from worked example

- *Major Effects* (ME).

The satisfied preconditions of a method refer to those meta-level conditions that must be *maintained* by the current method, such that the next method in the sequence can be applied. Therefore, these are a *subset* of the preconditions for the next method.

The major effects of a method refer to certain meta-level conditions that must be *achieved* by the method, such that the next method can be applied. These are also a subset (distinct from the major effects) of the preconditions for the next method.

Together, S and ME for each method provide a description of the *contribution* of the current method to the application of the next method in the sequence.

4.3.1 Learning Schema Methods

Figure 4–2 shows the satisfied preconditions and major effects for each method involved in the worked example of figure 4–1. These two pieces of knowledge, together with the name of the methods and the order in which they are applied form the main *body* of the newly learnt schema method [3].

The schema method *summarises* the worked example and provides a strategy for solving an equation which matches the preconditions of the first method in the sequence. However, even if a new problem matches these preconditions, applying the schema method does not *guarantee* a solution of the new problem.

Take, for instance, the worked example in figure 4–1. The sequence of methods succeeds because the equation 4.7 is of the form

$$a \cdot \cos(p) + b \cdot \cos(q) + a \cdot \cos(r) = 0 \tag{4.8}$$

---

[3]Other information contained in figure 4–2 comprises a record of the generating equation for the worked example and the unknown variable.

where $q = (p + r)/2$.

The schema method shown in figure 4–2 does not represent these constraints fully. Thus, LP may apply the schema method to any equation which satisfies the preconditions: that the left-hand side is a sum, the right-hand side is zero and there are multiple occurrences of the unknown. This over-generality of the applicability of the schema method falls foul when one of the methods in the sequence does not apply. In the worked example in figure 4–1, this could occur if, after the application of the cosine rule, there were no common subterms, such that the next method *factorisation preparation* could not be applied. The inability to capture the constraints of the equation 4.8, *ie* $q = (p + r)/2$ can result in failure to fully apply the schema method, in its current form.

However, knowledge about the contribution of each method, in the form of S and ME, provides the right sort of information for *patching* the application of the schema.

If a particular method, M, cannot be applied directly, then another method is chosen which achieves the same major effects as M and maintains its satisfied preconditions. If this is not possible, then another method is chosen, in the hope that the method M can then be applied. This method must not undo any of the preconditions of M that are already satisfied [4].

Take, for instance, a variation of equation 4.7

$$\cos(x) + 2 \cdot \sin(2 \cdot x) + \cos(3 \cdot x) = 0 \tag{4.9}$$

where one of the cosine terms is replaced by a sine term. Although, the cosine rule may be applied, it does not produce the common subterms required for the next method to apply. Thus, it must either be replaced by another method, or another method must be applied additionally to achieve the required major effect. In this case, by applying another method which reduces $\sin(2 \cdot x)$ to the following product of trigonometric terms

---

[4]The process of choosing alternative methods is not discussed here, but involved a look-up table based on which methods achieve which conditions.

$$2 \cdot \sin(x) \cdot \cos(x)$$

common subterms are now present for the next method to apply.

4.3.2 Learning New Methods

In LP, if a method is not known, then the satisfied preconditions and major effects for that method may be used to determine the specifications of the missing method. They are used in the following manner. Since, S refers to those conditions that must be maintained for the method such that the next method in the sequence is applicable, these must certainly be part of the preconditions for the unknown method. Since, ME refers to those conditions produced as a result of the method, such that the next method is applicable, then these must be part of the postcondition of the missing method. Additionally, since the conditions, S, are maintained by the missing method, then they must also be part of the postconditions.

Thus, for LP, the preconditions and postconditions of the missing method are provided by the following:

$$\textit{Preconditions} = \text{S}$$
$$\textit{Postconditions} = \text{S} + \text{ME}$$

An example of learning a new method can be seen in figures 4–1 and 4–2. The first method represented by the cosine rule is not known to LP and so a new method is learnt by determining its preconditions and postconditions by the above process. The resulting preconditions and postconditions are the following

Preconditions	rhs-zero(Eqn1)
	lhs-sum(X,Eqn1)
	multiple-occ(X,Eqn1)
Postconditions	rhs-zero(Eqn1)
	lhs-sum(X,Eqn1)
	multiple-occ(X,Eqn1)
	common-subterms(X,Eqn2)

Note that although these properties are valid for the worked example, they are not valid, in general. In particular, as we have seen in the previous section, the cosine rule does not always produce common subterms. Also, if the left-hand side of the equation contained only the sum or difference of cosine terms, then after applying the rule it would contain a product of cosine terms. Thus, the specifications that are determined with the aid of S and ME are the best that can be achieved, without further information.

4.4 Description of Precondition Analysis

This section provides a comprehensive description of the workings of precondition analysis within the LP program and a partial reconstruction of LP. This partial reconstruction has been implemented in PROLOG. The reconstruction contains only those parts of LP that are required to provide an explanation of precondition analysis. Other non essential details are avoided. The same can be said for the choice of an abstract example to describe the reconstruction. Avoiding unnecessary details permits stress to be placed on the relevant issues for machine learning.

4.4.1 How does LP work?

LP has four stages:

- Translate the problem from object-level to meta-level terms,

- Identify the application of existing methods,
- Apply precondition analysis to find S and ME, and determining any new methods
- Construct a new schema method

Translation of problem to meta-level terms

LP translates the state of the equation from object level terms into meta-level terms at each stage in the solution. The meta-level terms describe syntactic properties of the equation as explained in section 4.2.1. They describe the conditions under which a particular set of operations, characterised by the LP method, have proven successful in reaching the solution. There are a fixed number of these terms, considered sufficient for describing most of the required states for solving certain algebraic equations. New meta-level terms are not learned by LP.

Identification of methods

LP identifies the application of a known LP method whenever the preconditions and postconditions of the method match the meta-level description of the steps before and after the application of an rewrite rule which belongs to the method's own set of rewrite rules. Otherwise, no known method accounts for the step.

Application of precondition analysis

LP now applies the precondition analysis technique. This starts from the solution state and works its way back to the initial problem state. At each step, the aim is to find the reason for that method, such that the next method in the sequence can be applied. This is relatively straightforward for steps where methods have already been identified. But when no known method accounts for the step, LP has two ways to proceed:

- learn a new method whose preconditions and postconditions are based on the reasons for this method at this step in the solution, or else,

- if the method is known but lacks the particular rewrite rule for this step, it adds the new rule to the set of rules associated with the method [5]

An example of the former is when the step involves a sine or cosine rule, and there is no known method which accounts for the application of these rules. A new method can be learned for replacing a product of sines and cosines with a sum or difference of sine/cosine terms. The worked example in figure 4–1 shows how the cosine rule fills the gap in the solution.

An example of the latter, would be if the step involved the cosine rule, but an existing method only comprised the sine rule. Provided the preconditions and postconditions of the schema method matched the initial problem state, then the cosine rule, which reflects the change in state before and after the step, may be added to the set of rewrite rules associated with the method.

Construction of schema method

In the final stage, a schema method is constructed from the knowledge gained in the previous stages. This involves packaging all the methods required for the solution, including any newly learned methods, into a sequence of methods, where the order is vitally important. Each method in the sequence, has associated with it a name and its S and ME.

4.4.2 Partial Reconstruction of LP

The partial reconstruction described in this thesis is restricted to dealing only with some *abstract* meta-level description of a problem. These meta-level terms

---

[5]The new rule is acquired by referring to the object-level states before and after the step, and determining the lhs and rhs of a rewrite rule which reflects the transition between the two states.

are meant to describe properties of the state of the problem that are used to control the application of methods. There is no characterisation of the object level description of the problem or of the object-level problem solving operators. The aim is to show how precondition analysis manipulates the meta-level terms to provide explanations for each method in the solution to an problem characterised by some abstract meta-level terms.

For the purposes of this discussion, PA is described alongside an example problem. This example is used later on in the thesis to introduce major extensions to precondition analysis. The example is described below.

The Example

The example is represented by a transition of states from the initial problem state, $[a_1..a_n, b, c]$ to the solution or goal state, $[a_1..a_n, h]$. The progression of the state transitions is represented by the linearly-ordered list,

$$[[a_1..a_n, b, c], [a_1..a_n, d, e], [a_1..a_n, d, f], [a_1..a_n, d, g], [a_1..a_n, h]]$$

The translation process to the meta-level description of the solution is not dealt with in this discussion. The goal state is reached when the meta-level representation of the problem contains the meta-level term, h. Note that the other meta-level terms, $a_1..a_n$, in the final representation of the problem, $[a_1..a_n, h]$, are not needed in the solution, since they are not required by any of the methods. The meta-level terms $a_1..a_n$ represent those terms which are irrelevant to the solution.

Since there is no structural knowledge at the object-level, there are no rewrite rules to deal with. The methods are described by their specifications, *ie* the preconditions and postconditions. The identification of the methods now only requires the meta-level descriptions of the steps to match the preconditions and postconditions of known methods.

Let us assume that the known methods have been identified for each step in the worked example. The resulting representation of the solution after iden-

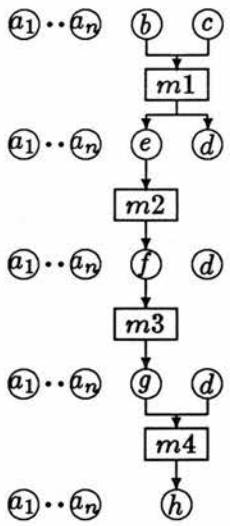


Figure 4–3: Representation of the solution after identifying methods

tifying the methods required to solve the problem is shown in fig 4–3 [6]. For the purposes of this example, all the methods required to solve the problem are already known.

The next stage involves the precondition analysis, where the idea is to determine the reasons for applying methods at each stage. This means determining the major effects and satisfied preconditions for each method required in the solution.

In LP it is assumed that the method which achieves the solution state is a key method [7]. Precondition analysis starts at a method which is *not* a key method and progresses back to the initial state.

---

[6]The circles with the arrows entering the boxes (methods) represent the preconditions and those with the arrows leaving the boxes the postconditions of that method.

[7]A key method is one which normally terminates a solution, *eg* the *isolation* method. Key methods do not require reasons for their application, since they lead directly to a solution. Thus, there is no need to determine their major effects or satisfied preconditions.

In the case of the solution in figure 4–3, precondition analysis starts from $m3$, assuming that $m4$ is a key method. To find S, look at the preconditions for the next method, $m4$, which are $[d, g]$ and determine which of these were achieved at the state to which $m3$ has been applied, $[a_1..a_n, d, f]$. S is the intersection of these two lists:

$$S = [a_1..a_n, d, f] \cap [d, g] = [d]$$

To find ME, take the satisfied preconditions S and subtract them from the preconditions of the next method, $m4$. Thus:

$$ME = [d, g] - [d] = [g]$$

This process is continued for the rest of the solution until the initial method is reached.

However, if the next method back in the sequence is one which was not known to LP, then it is still possible to determine its satisfied preconditions and major effects, since these do not rely on knowing its preconditions and postconditions. Without the preconditions of the unknown method, the precondition analysis stage cannot continue any further back through the solution. So it is at this point that they are determined, based on S and ME. Although, LP assumes the preconditions to be the same as S and the postconditions as the union of S and ME, a slight modification for determining the preconditions provides more accurate terms. By representing the preconditions by the union of S and the difference between the problem states before and after the step, LP would able to capture more representative preconditions.

Take, for instance, the method, $m2$. If this was unknown, LP would determine the that it had no preconditions, because S is empty. However, by considering the difference between the two problem states, *ie* $[a_1..a_n, d, e] - [a_1..a_n, d, f]$, the determined preconditions would be the set, $[e]$, which is more accurate.

The final stage consists of packaging the results of previous stages into a schema method as described in section 4.4.1. The structure of the schema method is represented in table 4–1.

Schema Method			
HEAD : $[b, c]$			
BODY :	*Method*	*S*	*ME*
	$m1$	[]	$[e]$
	$m2$	[]	$[f]$
	$m3$	$[d]$	$[g]$
	$m4$	–	–

Table 4–1: Structure of the Schema Method

4.5 Conclusions

Precondition analysis should be considered as the starting point from which to extend the EBL approach for representing and learning more control knowledge. The case for choosing precondition analysis rather than other EBL techniques discussed in chapter 2 relies on the following factors

- representing and learning the control knowledge at the meta-level provides not only more general terms with which to describe the control knowledge but also improves the search for problem solving.

- the learning process does not require all the methods to be known by the learning program. The specifications of unknown methods can be determined by manipulating knowledge about the specifications of the other known methods in the sequence.

In the next chapter, some problems with the current version of precondition analysis will be discussed and solutions presented which extend the current EBL approaches to learning more expressive control knowledge. The problems relate

to the limitations of control knowledge learned with LP. The reliance on S and ME, which only provide local information about each method, means that LP is not able to learn more strategic control knowledge. In particular, LP cannot adequately learn about either the applicability of the various stages of the schema method or the overall contribution of each method within the schema method. In addition, LP is also limited to learning about solutions involving sequences of methods, and not more complex networks of methods, such as trees or graphs.

These issues are addressed in the next chapter.

Summary of Chapter

In this chapter the following has been presented and discussed

- The role of meta-level knowledge and inference in both problem solving and machine learning has been discussed in the context of examples from the domain of algebraic problem solving

- The rational reconstruction of precondition analysis has shown how the technique can manipulate the description of the state of the problem to provide the necessary control information for learning and executing a representation of the solution. The technique was applied to an abstract example and a discussion of workings of the technique were elevated to that abstract level.

- Precondition Analysis contributes to two tasks:

  - *learning schema methods*, by finding the reasons for applying each method at each step of the problem solution, and

  - *learning new methods*, by determining the pre- and postconditions for the new method.

- Two pieces of information are deduced for each method involved in the solution to the problem, the major effects (ME) and the satisfied preconditions (S). Together they provide the *necessary* reasons for applying the current method in the sequence.

Chapter 5

An Extended EBL Approach to Learning Control Knowledge

5.1 Introduction

Chapter 2 has provided a survey of the various sorts of control knowledge that have been dealt with in AI from the perspective of both representational and learning issues. Chapter 4 has presented a case for developing the existing precondition analysis technique and for representing and learning the control knowledge at the meta-level.

Most enhancements to the EBL approach have involved improvements either to the performance of associated problem solving systems [Keller 87,Minton 87] or to the speed of EBL techniques [Prieditis 87,Mooney 86,KedarCabelli 87]. Important though these considerations are, the main objective of this thesis is in improving the *expressibility* and *power* of the EBL approach, particularly, for learning control knowledge.

The precondition analysis technique has shown its success in learning schema methods and new methods, by manipulating terms in the meta-language. The meta-language provides a good means of describing control knowledge, because it allows more general properties of the state of the problem to be represented.

Thus, the precondition analysis technique provides a good starting point from which to improve the expressibility of the EBL approach.

5.1.1 Motivation for extending Precondition Analysis

The precondition analysis technique, in its current form, provides a good record of how a particular method contributes to the application of the next method in the plan sequence. The other learning techniques, discussed in chapter 2, stress a combination of this knowledge and some of the other sorts of control knowledge.

Most of the previous EBL techniques manipulate knowledge at the object-level. Attempting to improve the expressibility of such techniques is inherently more difficult because of the restriction to one level of control knowledge. This is not the case, with the precondition analysis technique.

For precondition analysis, improvements to its expressibility can be achieved in two ways:

- by improving the form and content of the meta-language
- by extending the manipulation and use of these meta-level terms

The first option is the objective of research currently being pursued by other members of the Edinburgh Mathematical Reasoning Group. A combination of the first and second options is the approach adopted within this thesis, with my contribution provided mostly by extending the manipulation and use of meta-level terms.

The aim is to extend the precondition analysis technique by adding some of the facilities of the other EBL techniques. In this way, a new extended EBL approach should capture most of the different types of control knowledge mentioned in chapter 2.

5.1.2 Outline of the chapter

Section 5.2 identifies particular problems with the current precondition analysis technique. These are discussed in the context of the abstract example presented in chapter 4. The solutions to these problems address the central issues of improving the expressibility of the precondition analysis technique and involve the incorporation of the other EBL techniques.

Section 5.3 continues this discussion and presents the extended EBL approach. This incorporates the techniques involved in the previous solutions and is implemented in a program, Extended-EBL. Section 5.3.1 describes the application of Extended-EBL to a goal reduction problem. The benefits and limitations of Extended-EBL are discussed in the context of this example, in section 5.3.2.

Section 5.4 discusses the related work and section 5.5 presents some conclusions.

5.2 Problems and Solutions

The precondition analysis technique has proven very useful for the domain in which it was developed, *viz* algebraic equation solving. Most of the solutions to algebraic equations involve applying a sequence of operators (methods) to the initial state of the equation and transforming this state into the solution state. Precondition analysis works well with linear sequences of operators and captures valuable information about the contribution of operators within the sequence. However, the survey in chapter 2 shows how important the other types of control knowledge are for generating *good* plans.

The intention within this section is to identify problems with the current precondition analysis technique and to provide solutions to these problems. The aim of the section is to address the issue of extending the EBL approach by learning more expressive control knowledge.

5.2.1 Problems with Precondition Analysis

Three problems are addressed in this section [1]

- inability to represent spanning conditions
- better specifications for plans
- representing tree structures in plans

The problems represent areas which were not originally of prime concern for the LP program, but they show a lack of expressibility of the plans learnt with the precondition analysis technique, especially for more complex problem solving tasks [2].

Spanning Conditions

In LP, representing the explicit reasons for applying operators at each step in the schema proves to be very useful in schema execution. These reasons provided a flexibility in executing the schema. They provide knowledge for *patching* the schema when the next operator in the sequence cannot be applied. The patching process in LP comprises the following algorithm:

1. If a particular operator in the schema cannot be applied directly, then another operator is chosen which achieves the same major effects and maintains the satisfied preconditions.

---

[1] Control knowledge representing the utility of plans by a cost/benefit analysis is not dealt with in the rest of the thesis. In fact, it was only added to the survey chapter 2 for completeness. It is still a very recent area for machine learning and poses several difficult problems. However, doing justice to this area within this thesis would require a considerable amount of time and effort, and would detract from the existing contributions within the thesis.

[2] These problems were first identified and reported in [Desimone 87].

Schema Method			
HEAD : $[b, c]$			
BODY :	*Method*	*S*	*ME*
	$m1$	[]	$[e]$
	$m2$	[]	$[f]$
	$m3$	$[d]$	$[g]$
	$m4$	–	–

Table 5–1: Structure of the Schema Method

2. If this is not possible, then a operator is chosen which does not undo any already satisfied preconditions.

Such explicit reasons provide local information about the effect one operator has in providing the necessary preconditions for the next operator in the sequence to be applied. However, in many domains, operators often produce effects or conditions which aid the application of many other operators. These other operators may occur much later on in the schema and need not be restricted to the next operator in the sequence [3]. Such conditions which span more than one operator are called *spanning conditions.*

The example presented in chapter 4 provides a good context for this problem and is shown in figure 4–3 and table 5–1. Take, for instance, the operator $m1$ shown in table 5–1. The operator has two postconditions, d and e. The condition e is the major effect of $m1$, because it permits the next operator $m2$ to be applied. But the condition d is required for the operator $m4$ to apply later on in the schema. Explicit reasoning about the use of d is *not* recorded by

[3]This is certainly the case in many of the problems dealt with in the planning literature

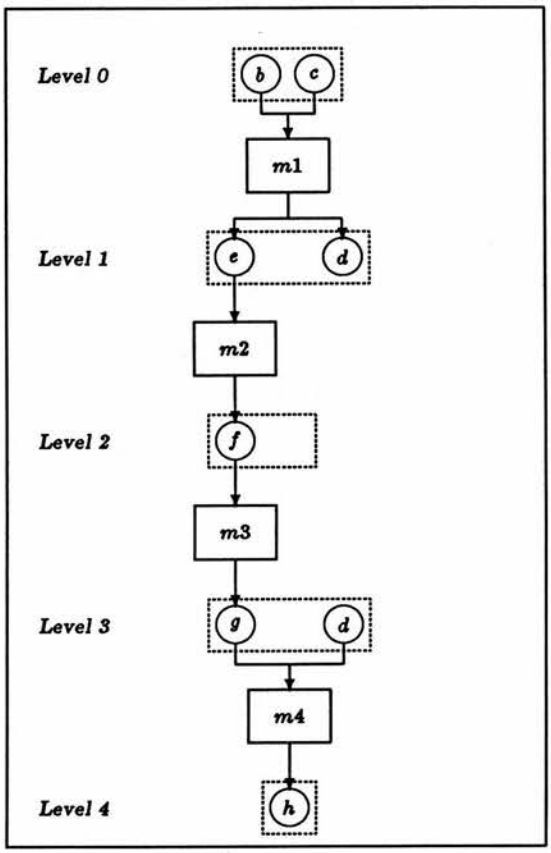


Figure 5–1: Representation of the learned plan after precondition analysis

precondition analysis. Why is this? Because it really only deals with the local effects of meta-level conditions from one operator to the next, and not further on. This is certainly a deficiency in the technique.

Why is this a problem? Suppose that in executing the schema, $m1$ cannot be applied, even though the preconditions, b and c are both present. This could arise if the preconditions for $m1$ were not complete, because they were learned from a more specific example, such that b and c are not sufficient to specify the preconditions of $m1$ for the more general case, *eg* another precondition, a_j, is required. However, another operator, $m*$, can be applied, but $m*$ has only the postcondition e, not d. Although, the schema can proceed through the sequence of operators, it eventually fails when it reaches operators $m3$ and $m4$, because the condition d is required. Consider the wasted time if the chain between these operators is very large.

Section 5.2.2 deals with this issue and proposes an enhancement to the existing precondition analysis technique.

Specifications for plans

The precondition analysis technique has a tendency to *over-generalise* the applicability of the learned schema, *ie* its preconditions are too general. This is because the specifications for the schema are represented by the preconditions of the initial operator only.

However, it is often the case that conditions which occur in the initial problem state are not relevant until much later on in the solution to the problem. If these conditions are not captured in the learned schema, as preconditions to the entire sequence involved in the schema, then the schema may fail when it is applied to another problem state, because it is too general for the schema.

Take for instance, a similar example to that shown in chapter 4, except that one of the conditions, a_k, is a precondition to the operator, $m4$. The learned schema provided by precondition analysis is represented in figure 5–2.

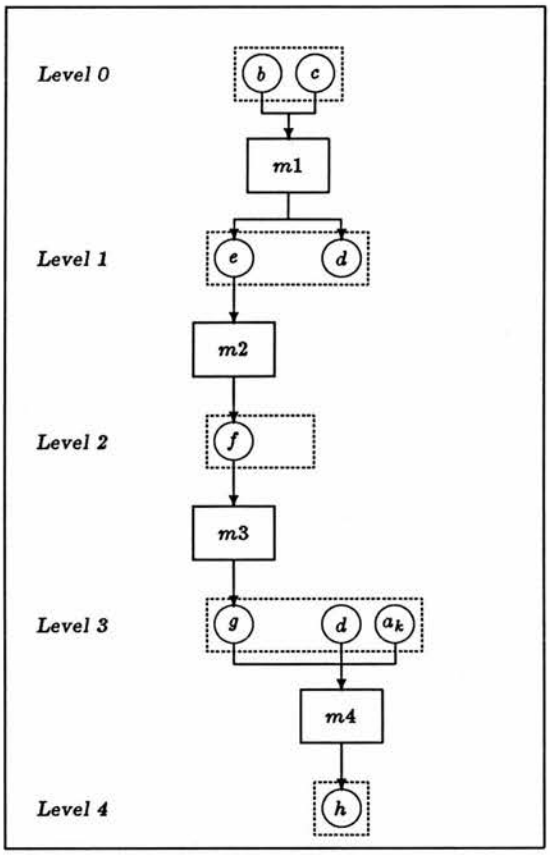


Figure 5–2: Representation of the learned plan after precondition analysis

The specification for the learned schema is provided by the preconditions, b and c.

When the schema is applied to a problem state, which matches these preconditions, but does not contain the condition a_k, it eventually fails. The operator $m4$ cannot be applied because one of its preconditions is not present in the current problem state. This failure is needless and is due to the over-generality of the specifications of the learned schema.

The solution involves including a_k as one of the preconditions to the entire sequence. This notion of preconditions to a sequence of operators, representing the specification for a plan, is not new. Planning research deals with this all the time. Even within machine learning this notion is the prime motivation

for the back propagation technique. These issues are pursued in more detail in section 5.2.3.

Plans as trees

The precondition analysis technique was originally developed for the domain of algebraic problem solving. In that domain, the solutions to algebraic equations tend to be displayed as linear sequence of transformations of the equations, progressing towards solutions. This is not quite the case, because of equations with several factors to solve. Thus, even in the domain of algebraic equation solving, not all solutions consist entirely of a linear sequence of operators.

Take the case of an equation with several values for the unknown variable, especially equations with trigonometric functions. The equation is normally factorised and the solutions determined for each factor.

$$2\cos 2x(\cos x + 1) = 0$$

has two solutions $x = 90n_1 + 45$ from $\cos 2x = 0$, and $x = 180(2n_2 + 1)$ from $\cos x + 1 = 0$. There is no ordering on which solution for x is performed first, so that the solutions could be determined in parallel. In LP, they are performed one after the other, in a linear sequence.

However, the rationale behind precondition analysis and the LP program assumes that the examples presented to the learning system and, thus, the schemata learned, involve a linear representation of the sequence of operators required to solve the equation [4]. Unfortunately, this is not the case with problem solving in general. Often problem solving is performed by a goal reduction process with several sub-goals to be achieved at each goal reduction step.

---

[4]In the planning literature this is called the *linearity assumption*

Therefore, rather than adopting a linearly-ordered list for denoting the application of the operators in a schema [5], a tree-like representation is better suited to representing the structure of a schema, which reflects the control structure for a goal reduction problem.

The intention in section 5.2.4 is to develop this idea further and to add this representational facility to the precondition analysis technique.

5.2.2 Improving Links between Operators within Plans

This section discusses an extension to precondition analysis that solves some of the problems with spanning conditions by providing a better representation of the entire contribution of each operator in the plan.

Rather than dealing only with a subset of postconditions of a particular operator *ie* the major effects and satisfied preconditions, the intention is to record knowledge about the effects of *all* of the postconditions of a operator, not just some of them. This involves recording *how* the operator affects not only the next operator in the sequence, but other operators later on in the sequence. Thus, it directly tackles the problem discussed in section 5.2.1.

It also increases what might be termed, *plan knowledge*, by emphasising control knowledge about the *entire* plan, rather than just the relevance of one operator to the next in the sequence. This is a radical change from LP where the use of control knowledge, in particular, the major effects and satisfied preconditions, contributes mostly to learning the specifications of new operators, rather than for learning plan knowledge. These ideas are evaluated in more depth in section 5.3 in the context of a concrete example.

In order to achieve this representation of plan knowledge, the notion of a *dependency graph* is required. It is not a new idea. In fact, it has been used

---

[5] As is the case with both the version of precondition analysis in LP and the extension described in section 5.2.1

in many other areas in AI research and, indeed, within the machine learning community [Fikes 72,Mooney 85,Benjamin 87].

The dependency graph provides a record of links between one operator and another in the sequence, although they may be spread over the entire sequence. These links are between the preconditions of one operator and the postconditions of another. Instead of attaching the major effects and satisfied preconditions to each operator in the sequence, a dependency graph is generated consisting of the following:

- *preconditions* – together with links to operators earlier in the sequence which produced the meta-level conditions as their postconditions,

- *postconditions* – together with links to operators later on in the sequence for which they provide preconditions.

Consider the addition of the dependency graph to the example presented in the previous chapter 4. Note that, in the example, one of the postconditions of the operator, $m1$ is required as a precondition to the operator, $m4$, later on in the sequence. The use of dependency links between such condition can bridge gaps which have been left in the overall plan structure. Figure 5–3 shows the representation of the learned plan (schema method) together with the necessary dependency links. The dashed boxes denote that the meta-level properties within them refer to the same object-level state of the problem. There is some repetition of these meta-level properties. This is done to emphasise the dependency links between the postconditions of one operator and the preconditions of another.

The process of generating these dependency links can be performed either in the forward or backward direction. The former involves starting from the first operator in the sequence and gradually filling in the dependency links until the last operator is reached. The latter involves starting from the last operator in the sequence and working back to the first, as is the case in LP.

The backward-directed process is preferred for several reasons, some which are explained now, and some of which are explained later on in the chapter.

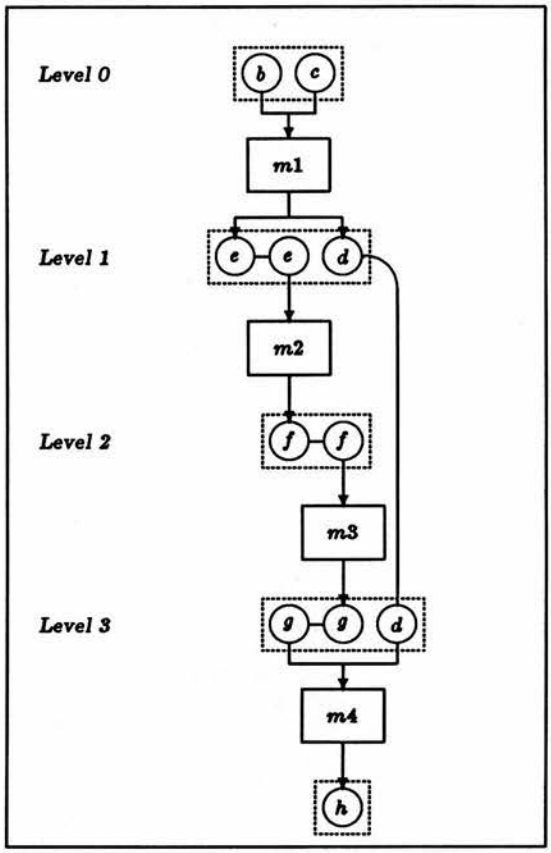


Figure 5–3: Representation of the learned plan including dependency links

Although, performing this process in the forward direction appeals because of its simplicity and naturalness, it does require that *all* the operators in the sequence are known before generating the dependency links. If just one operator is missing, then the process must wait until the operator has been learnt before continuing. Since learning the new operator involves reasoning backwards from the operator after the missing one, it seems more practical to have the entire process performed backwards from the last operator.

The process of constructing the dependency graph is described as follows:

1. The process begins at the last operator in the sequence.

2. If the operator is known then proceed to 3, otherwise a new operator must be learnt, in order to provide the necessary preconditions and postconditions for the missing operator.

3. Taking each precondition for the current operator, a search is made back in the sequence of operators for the one which produced this precondition. This is achieved by attempting to match the particular precondition with the postconditions of previous operators. The links are recorded for each precondition and postcondition, resulting in both the links to a operator and from another.

4. Proceed to the previous operator in the sequence and go back to 2.

The result of applying this process to the example from the previous chapter 4 is shown in figure 5–3. Table 5–2 shows the revised structure of the plan. Note that figure 5–3 includes the link between the condition d which is a postcondition for $m1$ and a precondition for $m4$. Now for each operator in the sequence, there is an *explanation* of how it fits into the entire sequence. There is a record of how each of the postconditions of operators in the sequence contribute to the schema. The sequence of operators is still represented by a linearly-ordered list, but the dependency links may span several operators.

Plan			
HEAD : $[b, c]$			
BODY :	*Operators*	*Preconditions*	*Postconditions*
	$m1$	$precond(b, -)$ $precond(c, -)$	$postcond(d, m4)$ $postcond(e, m2)$
	$m2$	$precond(e, m1)$	$postcond(f, m3)$
	$m3$	$precond(f, m2)$	$postcond(g, m4)$
	$m4$	$precond(d, m1)$ $precond(g, m3)$	$postcond(h, -)$

Table 5–2: Structure of the Plan with dependency links

5.2.3 Learning Better Preconditions for Plans

This section addresses the problem discussed previously about providing preconditions for a sequence of operators rather than just for a single operator. Such preconditions should capture those conditions which are required in the initial problem description for the particular sequence of operators to be applied successfully. Thus, together with the structure of the operators involved in the sequence and the dependency links between the preconditions and postconditions of each operator, a representation of the preconditions for the sequence of operators provides essential information for describing the applicability of the plan.

Take, for instance, the example shown in figure 5–2. The precondition, a_k, for the operator, $m4$ is an essential precondition for the plan described by the sequence of operators, $m1 - m4$. It could have been produced by one of the operators in the sequence, in which case the condition occurs as the result of that operator and a dependency link is recorded. On the other hand, if no

operator in the sequence has generated this condition, then it must have been present in the initial problem state.

By combining the generation of dependency links with a form of the back propagation technique, it is possible to determine the preconditions for each sub-sequence of the entire plan. This procedure is similar to that discussed in section 5.2.2 for generating the dependency graph and is presented below.

1. The procedure starts from the last operator in the sequence.

2. If the operator is known then proceed to 3, otherwise a new operator must be learnt, in order to provide the necessary preconditions and postconditions for the missing operator.

3. Taking each postcondition for the current operator, a match is attempted with the preconditions for the next stage of the plan, which includes the sequence of operators after the current operator. If a match is successful, then a dependency link is recorded. Otherwise, each precondition for the next stage which does *not* match one of the postconditions of the current operator is recorded as an *unmatched* precondition. If there are no operators following the current operator then there are no dependency links or unmatched preconditions.

4. These unmatched preconditions are then propagated back through the current operator and included with its own preconditions. Together these represent the preconditions for the sequence of operators which begin with the current operator.

5. Proceed to the previous operator in the sequence taking the preconditions for the next stage and go back to 2. Else if no previous operator, stop.

Eventually there should be dependency links between all the preconditions and postconditions involved in each operator, except for the preconditions for the entire plan. Figure 5–4 shows the result of applying this procedure to the example represented in figure 5–2. Note that there are two types of links in figure 5–4,

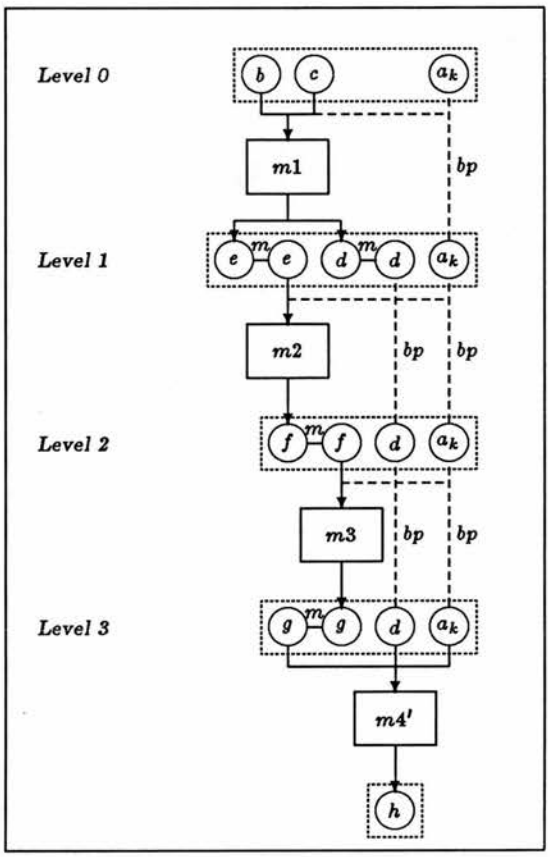


Figure 5–4: Representation of the learned plan with dependency links and back propagated preconditions

m and bp. The m links denote that the postconditions of one operator match the preconditions of another. The bp links, shown with the dotted lines, denote the back propagated preconditions. Together these shown the dependency links involved in the learned plan.

5.2.4 Representing Trees in Plans

This section addresses the issue of representing and learning more complex structures of operator sequences. A suitable example with which to discuss this problem involves a solution to a problem by a goal reduction process, as shown in figure 5–5. Instead of the solution being represented as a linear transformation

of states, it is represented as a tree whose leaves correspond to solved states [6] of sub-goals of the initial problem state, partly described here by the meta-level preconditions, $[b, c]$, of the first operator, $m1'$. Each application of a goal reduction operator, results in one or more sub-goals to be solved.

An operator is identified as having been applied in the solution, if its preconditions and postconditions match the meta-level description before and after the transformation of the problem step. However, for a goal reduction problem, the solution may contain more than one sub-goal for each problem reduction step. Since it is the application of each operator which reduces the goal to several sub-goals, this means that each operator may have several sets of postconditions associated with it. The identification of the operators involves traversing the tree in a *depth-first* manner, expanding first the left-most branch of each sub-tree. Since there is no preference in the ordering of the sub-goals, this choice is arbitrary [7].

In LP, the precondition analysis phase starts from the solved state and progresses back until all the major effects and satisfied preconditions had been determined for each operator back to the initial operator applied. However, because of the tree-like representation of the solution, it is unclear where to start the precondition analysis phase – since there are many solved states, represented by the leaves of the trees.

Thus, the revised process involves traversing the tree structure of identified operators in a depth-first manner [8], until no more operators exist in that branch and a leaf node is reached.

---

[6] For this problem, let us assume that a *solved* state must contain the condition, h.

[7] However, since the implementation of the technique is written in PROLOG, the depth-first expansion of trees is relatively simple to encode.

[8] As is the case for the operator identification phase.

Then, the rest of the revised learning algorithm, modified to take account of the representation of trees, is applied. The revised version of the learning algorithm is as follows

1. The procedure starts from the last operator in the sequence. This may be the last operator in a list of operators or the last operator in the left-most subtree, depending on the structure of the solution.

2. If the operator is known then proceed to 3, otherwise a new operator must be learnt, in order to provide the necessary preconditions and postconditions for the missing operator.

3. The matching process may involve several sets of postconditions depending on the number of sub-trees generated by the current operator. Assume that the following process applies for each postcondition set. Taking each postcondition for the current operator, a match is attempted with the preconditions for the next stage of the plan, which includes the sequence of operators after the current operator. If a match is successful, then a dependency link is recorded. Otherwise, each precondition for the next stage which does *not* match one of the postconditions of the current operator is recorded as an *unmatched* precondition. If there are no operators following the current operator then there are no dependency links or unmatched preconditions.

4. These unmatched preconditions are then propagated back through the current operator and included with its own preconditions. Together these represent the preconditions for the sequence of operators which begin with the current operator.

5. Proceed to the previous operator in the sequence taking the preconditions for the next stage and go back to 2. Else if no previous operator, stop.

The resulting plan learned from the goal reduction problem, shown in figure 5–5, is presented and discussed in the next section.

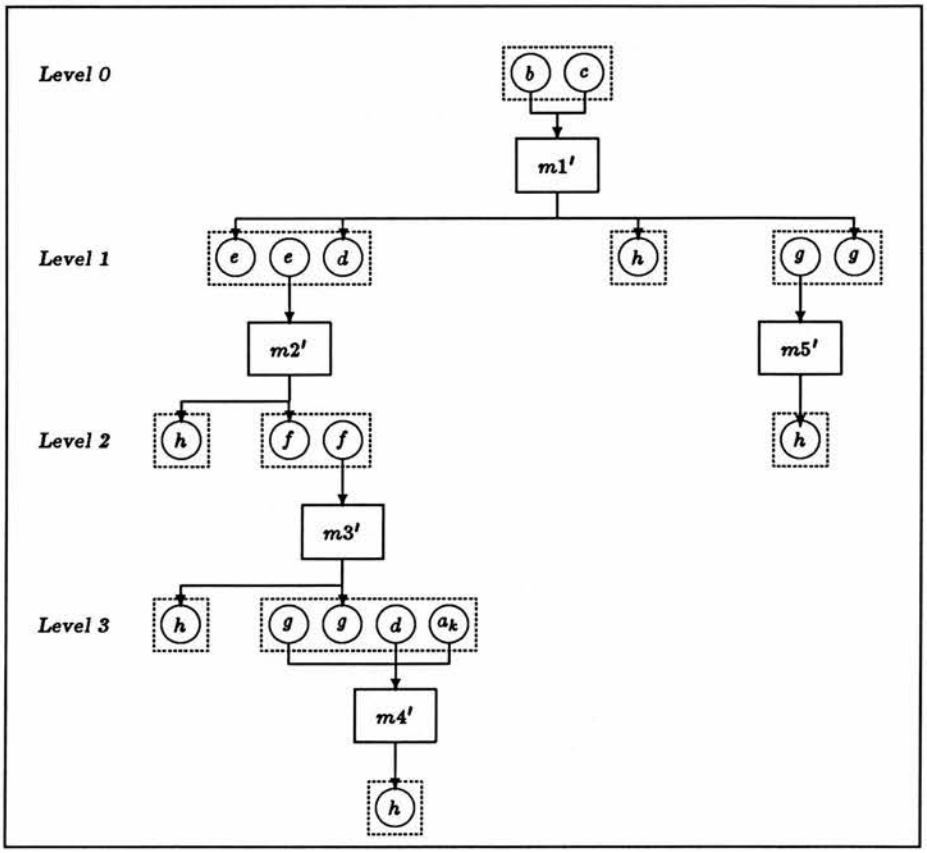


Figure 5–5: Representation of the operator tree structure

5.3 The Extended EBL Approach

The extended EBL approach is now discussed in the context of the example from the previous section. The example is rather abstract, but it does permit discussion of the contribution of the new extended EBL approach. It shows, in particular, how by combining the techniques of back propagation, with the use of a dependency graph and the original structure of the precondition analysis approach, most of the forms of control knowledge discussed in chapter 2 can be determined and used to construct a plan for solving certain problems.

The extended EBL approach has been implemented in PROLOG and the resulting program has been named Extended-EBL. Details about structure of the program are given in appendix B.

The example that is presented to the learning program, Extended-EBL is shown in figure 5–5. This figure shows the representation of the operators required to solve the problem. The process for identifying the operators which formed part of the LP program, from which the precondition analysis technique is derived, is not examined within this thesis. The operator identification phase involves matching the meta-level preconditions and postconditions of the operators respectively with the object-level descriptions of the problem states before and after the operators. In this case, the solution to the problem involves the application of a tree of operators, and so the solution can be considered to have been provided by a goal reduction process.

5.3.1 Resulting Plan

The aim of the learning system is to determine the relevant control knowledge for constructing the plan, *ie* the preconditions for each stage of the plan, the contribution of each operator and the explicit representation of the structure of the plan.

The application of the program has been displayed at various stages between figures 5–6 and 5–8, in order to show Extended-EBL, in action, and to represent the resulting learned plan more clearly.

The example is input to the program in the form of a tree of operators, which, in this case, involve the operators, $m1' - m5'$, denoted by the labelled boxes. All the operators involved in the example are already known to the system, so that no new operators are learned. The specifications for each operator are described by the labelled discs, where the preconditions to each operator are represented by the discs which enter the operator box, the postconditions are denoted by the discs which leave the box. These specifications are also known to system, so that, before any learning takes place, all the items of knowledge shown in figure 5–5 are known by Extended-EBL. The dotted boxes which enclose sets of disc represent the fact that the discs represent properties about a particular problem state.

The algorithm described in section 5.2.4 embodies the main procedures in Extended-EBL.

The learning procedure begins by exploring the tree structure of operators, attempting to find the preconditions for the next level of the plan. This involves taking the left-most branch of the sub-tree after the application of each operator and recursively calling the learning procedure. The recursion stops when a leaf node of the tree is reached, in which case there are no preconditions for the next level. The other sub-trees, which result after the application of the operator, are explored recursively by the learning procedure, until there are no more sub-trees for that level.

In this example, the first operator, $m1'$, has three sub-trees branching from it. Taking the left sub-tree at level 1, the next operator is $m2'$, which has two sub-trees following it. The left sub-tree at level 2 contains a leaf node, represented by the postcondition, h, which denotes a solved state. Since no more operators follow this branch of the tree, there are no precondition for the next level. The right sub-tree at level 2 leads to the operator, $m3'$, which also has two subsequent sub-trees. Again the left sub-tree leads to a leaf node and the right sub-tree leads

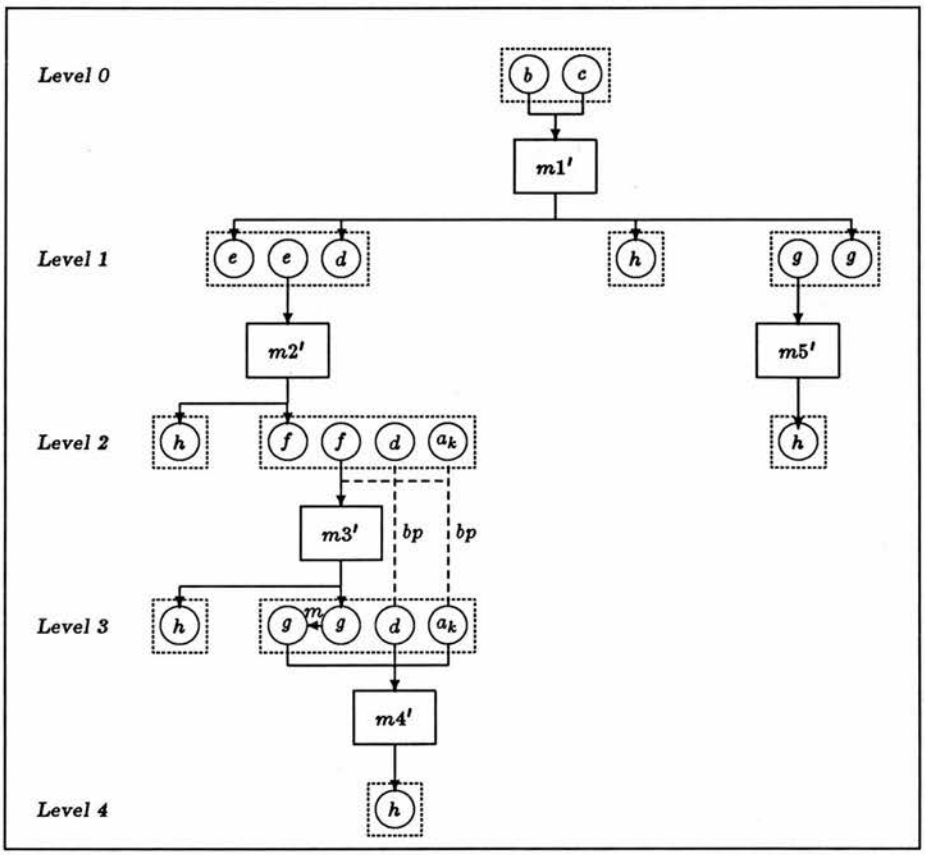


Figure 5–6: Representation of the Plan level 2

to the operator, $m4'$. This is the final operator in this part of the tree. There are no more operators following and only a single leaf node represented by the postcondition, h. The recursive process of exploring the sub-trees after each operator application ends at this point, since there are no more operators in the tree.

Now the next step in the learning procedure can take place [9]. This involves the matching process between the postcondition of the current operator and the preconditions for the next level of the plan. Since there are no preconditions for the next level of the plan, level 4, the matching process results in no matched or unmatched preconditions.

The next step of the learning procedure involves back propagating the unmatched preconditions and appending the resulting back propagated preconditions to the preconditions of the current operator, $m4'$. This results in the preconditions for the right sub-tree at level 3, which are, in this case, simply the preconditions of $m4'$.

Unravelling the recursion further means applying the matching process to the left and right sub-trees following the operator, $m3'$, at level 3 of the plan. There are no matched or unmatched preconditions for the left sub-tree. However, for the right sub-tree one of the preconditions, g, matches the postcondition, g of operator $m3'$.

The remaining preconditions are added to the set of unmatched preconditions and propagated back through the operator, $m3'$. Thus, the preconditions for the right sub-tree after operator, $m2'$, at level 2, comprise the back propagated preconditions, d and a_k, and the precondition, f, of the operator $m3'$. The representation of the plan at level 2 is shown in figure 5–6.

The recursion is unravelled further and the matching and back propagation processes are applied to the sub-trees following the operator, $m2'$, at level 2. The resulting plan for the left sub-tree after operator, $m1'$, at level 1 is shown in figure 5–7.

At this level of the tree, there are two further sub-trees after operator $m1'$ to explore. One leads immediately to a leaf node. The other leads to the application of the operator, $m5'$. The preconditions for the next level of the plan at level one

---

[9]The step for learning new operators is irrelevant since all the operators are known.

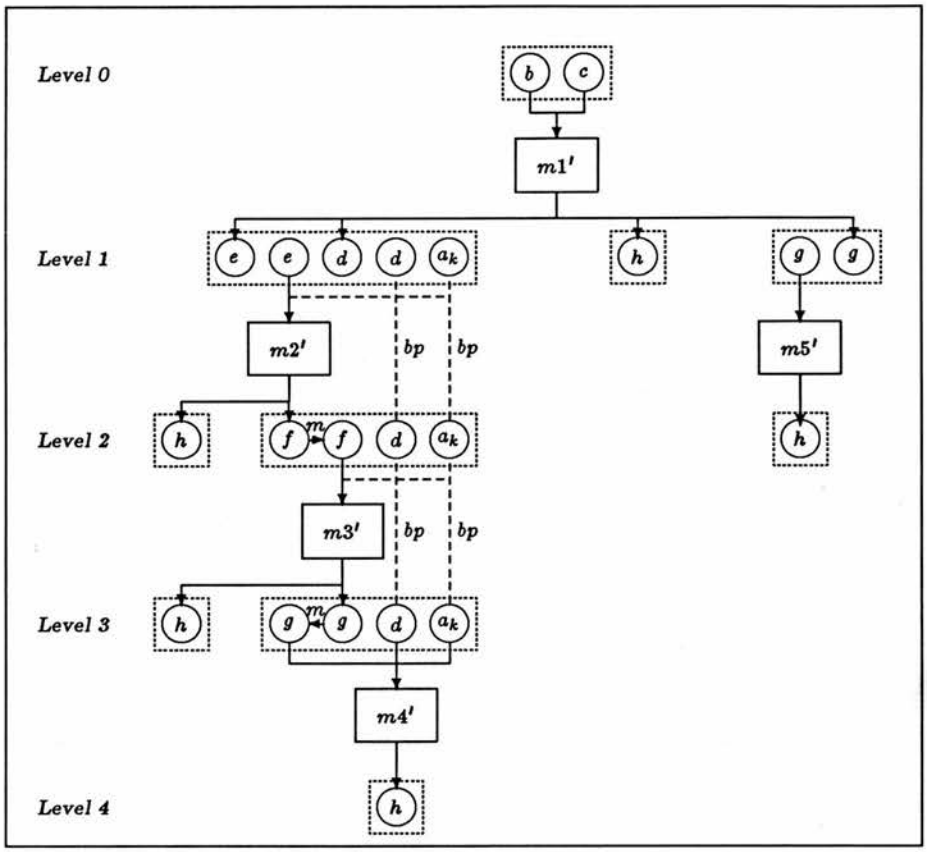


Figure 5–7: Representation of the Plan level 1

comprises three sets of precondition for each of the three sub-trees after operator, $m1'$. These are represented by the three sub-lists of the list $[[e, d, a_k], [], [g]]$.

Applying the matching process results in the matched preconditions, e and d for the left-most sub-tree and the precondition, g, for the right-most sub-tree. Since there is only one unmatched precondition, a_k, this is propagated back through operator, $m1'$. The resulting representation for the complete plan at level 0 is shown in figure 5–8.

The preconditions for the entire plan are represented by the set of preconditions, $[a_k, b, c]$. At each level of the plan, the preconditions required for the application of the next level of the plan are recorded. The contribution of each operator at each level of the plan is shown in figure 5–8 by the matching links, m, between postconditions of one operator and the preconditions of the next level. In addition, when the matched precondition is actually required for an operator which is applied much later on in the sequence, another link, bp, connects the matched precondition to the level of the plan at which the operator is applied and to the intermediate levels as a precondition for the next level.

For instance, the precondition, a_k, is required for the entire plan to succeed. However, it is not actually required until level 3 of the plan, when the operator $m4'$ is applied. Yet, it is recorded as a precondition for each level of the plan: level 0, level 1 level 2 and level 3.

5.3.2 Benefits and Limitations

Role of the dependency links

The dependency links provide the means of expanding the representation of a plan from a description of just the sequence of operators, which is input to the learning system, to include within the representation of the plan the various types of control knowledge mentioned in chapter 2, *ie*

- the contribution of each operator within the plan

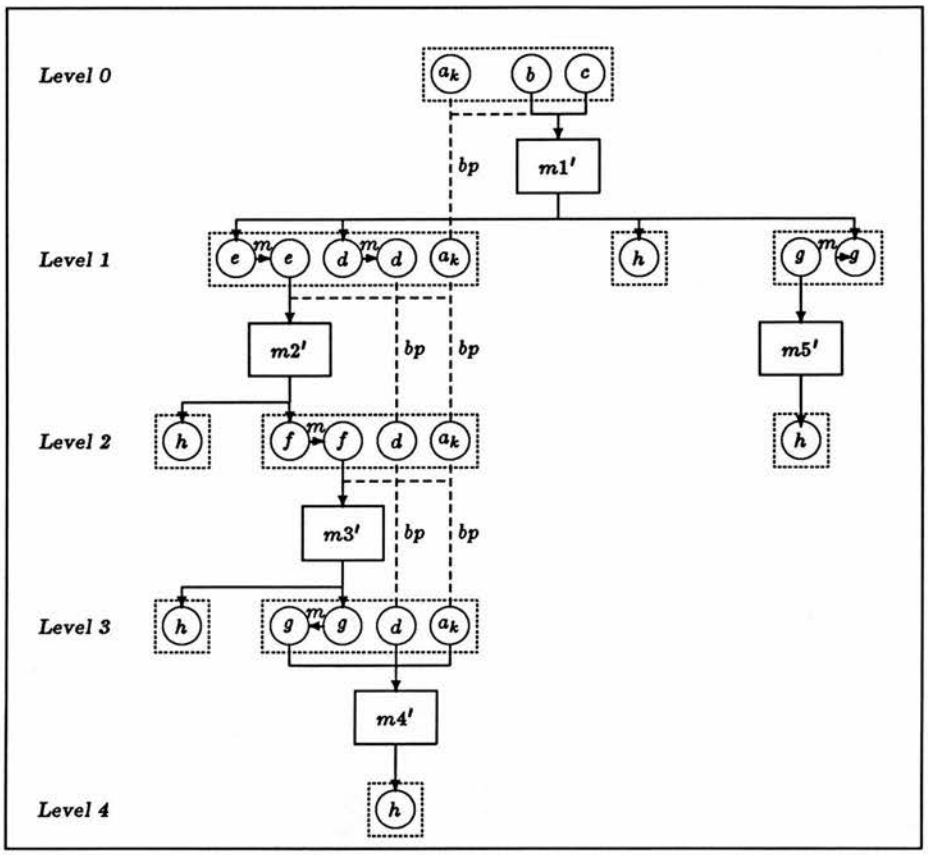


Figure 5–8: Representation of the complete plan

- representing the preconditions for the next level of the plan

- providing the explicit structure of the plan

The two types of links, *m* and *bp*, produced by the matching process and the back propagation processes respectively, described earlier in the chapter, provide the necessary information for recording the dependency links.

The dependency links are determined between the postconditions of one operator and the preconditions of another operator. These are recorded for every operator involved in the plan, in order to represent their contribution to the plan. They also provide the necessary links for describing the structure of the plan, whether it is a linear sequence of operators, a tree of some other structure such as a graph or lattice [10]. The dependency links provide useful knowledge not only for generating the plan, but also for dealing with failure in plan execution. This is dealt with next.

Dealing with failures in plan execution

Although, the plans acquired by Extended-EBL have been learnt from successful solutions to problems, they cannot be guaranteed to succeed on other examples that match the preconditions of the plan. There are several reasons for plan failure

1. missing terms in the descriptive language for the specifications

2. learnt or existing operators do not have complete specifications

3. conditions which should span operators are violated

---

[10]The latter two have not been shown in examples, but it is relatively easy to expand the learning algorithm to deal with a graph or lattice structure

However, the dependency links provide a lot of knowledge for dealing with plan failures.

Consider the situation where a learnt plan has been selected because its preconditions match the current problem to be solved. Let us assume, in this case, that the learnt plan is the one represented in figure 5–8. Assume that the first operator, $m1'$, has been successfully applied and operator, $m2'$, is ready to be applied to the left-most sub-tree. However, the operator, $m2'$, cannot be applied.

The failure to apply the operator may be due to one of the above reasons. An example of the first reason could be the fact that one of the preconditions or postconditions of the operator may be too general. It could be that the precondition, e, represent some property, such as the problem containing *trigonometric* terms, *ie* sin, cos or tan. However, the operator, $m2$, in fact, only applies to problems with sin and cos terms [11]. Thus, the operator, $m2'$, must be replaced with one that only deals with sin or cos terms, for the plan to succeed.

The second reason for plan failure is a variation of the first. Although the preconditions and postconditions of the operator may match the state of the problem, another precondition, a_n, is required for $m2'$ to be successfully applied. The original preconditions, which may have been learned previously from a more specific example, are not complete specifications for the operator, $m2'$. Again, the operator, $m2'$, must be replaced.

The third reason involves conditions which span operators. Assume that a_k is one of the preconditions of the learnt plan, but is required much later for the operator, $m4'$. During the spanning period it is assumed that the condition

---

[11] If the meta-language does not contain a term representing either sin or cos terms, then this could explain the choice of the more general term, *trigonometric*, for the operator, $m2'$.

remains true [12]. However, if the operator, $m2'$, invalidates the property described by the condition, a_k, then the plan will fail at operator, $m4'$. The operator, $m2'$ can either be replaced or another operator added to preserve or return the property to the problem description.

In each of the above three cases, the dependency links provide valuable information for 'patching' the plan. Knowledge about the contribution of the operator and how it fits into the structure of the plan is useful for either choosing another operator to replace the existing one or for adding or removing an operator to the existing structure.

Learning specifications of missing operators

In LP, the specifications of missing operators were inferred with the aid of the satisfied preconditions (S) and major effects (ME) for that operator. These could be determined, even though the operator was not known. Thus, the preconditions and postconditions of the missing operator were found by making use of the two pieces of information S and ME. It would be very simple to adopt the same process for learning the specifications of missing operators. However, the failing of LP and precondition analysis to identify spanning conditions and represent the entire contribution of each operator in the plan, prevents the ability to learn adequate specifications [13].

Even the enhancements incorporated in Extended-EBL do not allow for adequate representation of the relevant specifications. The problem is now discussed.

In LP, the preconditions and postconditions for the missing operator are determined in the following manner:

---

[12]This is generally termed as the *Modality Truth Criterion* (MTC), in the area of planning research.

[13]Indeed, the ability to learn such specifications is flawed, as has already been shown in chapter 4.

Preconditions = S

Postconditions = S + ME

Suggestion were made in chapter 4 on how to improve the representation of the preconditions by taking into account those conditions which do not exist, *ie* are *deleted*, by the missing operator. Thus,

Preconditions = S + D

where D stands for those conditions deleted by the operator.

However, the *local* nature of the conditions, represented by S and ME, results in an incomplete representation of the required specifications. This because S and ME represent only those conditions which refer to the interaction of adjacent operators. Thus, spanning conditions, which could also form part of these specifications, are not taken into account.

The problem lies in trying to associate the right spanning conditions with the right operator. For a single missing operator, those spanning conditions which do not link with any other operator could be considered in determining the specifications of the missing operator. However, if there are two missing operators, then deciding which spanning conditions should be associated with which operator is non-trivial. Potential solutions are presented in this thesis. These are deferred until later on in chapter 6, where discussions of the interaction between object-level and meta-level knowledge are expounded.

5.4 Related Work

MACROPS

The work on MACROPS [Fikes 72] in the mid 1970s resulted in a technique for learning and executing generalised robot plans. This involved identifying the role of each operator within the plan and representing the internal structure of the plan.

The dependency graph in Extended-EBL, provides, at least, as much information about the contribution of each operator to the entire plan and is also able to explicitly represent the internal structure of the plan. In addition, Extended-EBL has some ability to learn the specifications of missing operators.

In MACROPS the form of the plan is represented by a *triangle table*, a tabular format shaped as a lower triangular array. The rows and column of this array represent the operators of the plan and the contents of each row are the preconditions and postconditions of each operator. The dependency graph represents the same information in the form, naturally, of a *graph.*

However, the graph representation provide something the triangle table cannot, *viz* the ability to extend itself to the representation of plans that do not just involve a sequence of operators. The extension in section 5.2.4 shows how the dependency graph can represent the plan structure required for tree-like goal reduction problems [14].

LP/Precondition Analysis

The more recent research on LP and precondition analysis owes a lot to the original work on MACROPS, particularly for the representation of the contribution of operators within the learned plan.

The representation of the satisfied preconditions and major effects in LP is superceded in Extended-EBL by the use of the dependency links between *all* of the preconditions and postconditions of operators involved in the plan.

Although the schema method in LP does not fully capture the representation of a plan in the same manner as MACROPS and Extended-EBL, the precondition analysis technique does address the important area of learning the specifications of missing operators within the plan.

---

[14]The ability to represent this tree-like structure is invaluable for theorem proving, as is shown in chapters 3 and 6.

Enhancements to the process of learning such specifications have already been discussed in chapter 4. Further enhancements and more detailed exploration of the problems involved are dealt with later on in chapter 6. It shows how the use of the dependency links and interaction between object and meta-level knowledge can provide more complete specifications.

EBG/LEX2/PET

The three programs, EBG, LEX2 and PET are able to represent the most general preconditions for a sequence of operators such that the application of this sequence results in a solution. Each employs the same technique of back propagating constraints on the description of these preconditions.

Extended-EBL incorporates the constraint back propagation technique in order to achieve the necessary dependency links for spanning conditions and to represent the preconditions for each level of the plan. Chapters 3 and 6 show how the back propagation technique in Extended-EBL can be enhanced further. This involves representing explicitly the operator transformation, as is the case in PET with *relational models*, and using these to provide more control knowledge that is useful for learning the specifications of missing operators and for dealing with plan failure.

5.5 Conclusions

The extended EBL approach represented by the program Extended-EBL has shown how past efforts at learning control knowledge can be incorporated within a single program. The resulting program is able to capture most of the forms of control knowledge described in chapter 2.

The role of the dependency graph is central within Extended-EBL. The dependency graph provides a flexible way of representing the entire contribution of each operator and the internal structure of the plan. It provides as much

information as the MACROPS triangle table about the internal structure of the plan and contribution of its operators, and more.

Extended-EBL is able to represent the preconditions for each level of the plan. Furthermore, the structure of the plan is not restricted to a linear sequence of operators. More complex structures such as tree, graphs and lattices are realisable with the dependency graph. In addition, the Extended-EBL still has the ability to use the control knowledge it acquires for learning the specifications of missing operators and for providing useful information for patching plans that fail during execution.

Summary of the Chapter

In the chapter the following has been presented and discussed

- Some problems are identified with the existing precondition analysis technique. These problems mostly involve the inability to represent many of the previously described forms of control knowledge. The solutions to these problems involve
  - improving the links between operators
  - learning better preconditions for plans
  - representing trees in plans
- An Extended EBL approach is discussed and the new learning algorithm incorporated in a program, Extended-EBL. An example is presented to Extended-EBL, and the resulting learned plan is described and analysed. The benefits and limitations of the Extended EBL approach are discussed in the context of the example.
- Extended EBL is compared with related work in the area of EBL research. Its ability to represent many of the relevant forms of control knowledge is noted.

Chapter 6

The Role of the Meta-Language for Extended-EBL

6.1 Introduction

The extended EBL technique described in chapter 5 shows how the various types of control knowledge can be combined to form a more *comprehensive* plan for solving particular problems. The analysis of the example in chapter 5 at an abstract level permitted stress to be placed on the *use* of these types of control knowledge for learning plans. In particular,

- The contribution of each of the operators within the learned plan – by finding the dependencies between *all* the preconditions and postconditions of *every* operator involved in the solution to the problem.

- The representation of more complex networks of operators within the plan structure, such as a tree or graph structure, rather than just a simple *linear* sequence.

- Better specifications for each *stage* of the plan, rather than just the preconditions of the next operator in the network.

- The ability to learn the specifications of *missing* operators that may be added to the set of known operators

The resulting plans are more *general* because of the improved knowledge about the applicability of the operator network, particularly for each stage of the plan. The plans are also more *expressive* or *specific*, because of the explicit knowledge of the contribution and structure of the operator network. Thus, the emphasis was on how the use of such control knowledge provided a much more comprehensive description of the internal structure and applicability of the plan.

Because the level of discussion in chapter 5 treated the specifications of operators as propositions, the role of the meta-language within Extended-EBL was not discussed fully. However, the meta-language plays a vital role within Extended-EBL not only for representing the more general specifications of operators [1], but also for guiding the processes within the learning program.

Previous work on PRESS and LP [Sterling 82,Silver 85] has shown how a meta-level description language can be used for representing the specifications of operators and plans. Indeed, the PRESS project shows how the use of a meta-language makes the automation of algebraic equation problem solving much more effective. By performing the problem solving at the meta-level, the search for solutions is constrained to meta-level terms [Sterling 82].

With a meta-language it is easier to provide descriptions of the *properties* of problem states, rather than direct object-level descriptions of the state. Relying on an object-level language to provide the specifications of the operators could lead to a situation where a plan is not applicable because its specification does not quite match the problem state in object-level terms, although the properties of the states are the same. Representing the specifications at the meta-level

---

[1] Chapter 3 proposes a meta-theory, comprising meta-level preconditions, such as `contain(Formula,List_of_vars)`, and effects, such as `replace_all(Variable,Value,Old_formula,New_formula)`.

provide the plans with more general applicability. This is a very important consideration for learning plans.

Although, the object-language provides terms that can easily match the problem state, these are not general enough to describe the reasons for applying an operator. As we have already seen in LP, the meta-language is certainly able to cope with this task.

When it comes to applying Extended-EBL to any domain great effort is required in representing the form and content of this control knowledge, rather than just stressing how and where it fits in to the learned plans. Without the meta-language Extended-EBL is a mere *skeleton* upon which flesh must be added. In addition, it is shown within this chapter that Extended-EBL relies on the meta-language for enhancing the techniques for generating dependency links between operators, for back propagating preconditions and for learning the specification of missing operators.

The outline of the chapter is as follows.

Section 6.2 reviews the learning algorithm presented in the previous chapter and identifies where the meta-language plays its prominent role.

Section 6.3 describes how the addition of the meta-language complicates the the matching process required for generating dependency links, but ends up with a much stronger description of the contribution of the operators within the learned plan.

Section 6.4 discusses how the meta-language helps enhance the back propagation process such that it not only provides general preconditions for each level of the plan but that they are also coherent and consistent with the object-level descriptions of each level of the problem states.

Section 6.5 shows how the meta-language can be used to identify the specifications of missing operators, although with qualified success.

Section 6.6 discusses the benefits and limitations of Extended-EBL in the context of related work.

Finally, section 6.7 presents some conclusions about Extended-EBL, discussing the contribution it makes to the area of explanation-based learning.

6.2 The Learning Algorithm – A Review

This section shows where and how the meta-language affects the previous description of Extended-EBL. The overall structure of the learning algorithm is not changed too much. However, some of the sub-processes within the algorithm are affected considerably and require much more fleshing out. In addition, two more sub-processes are added to deal with the task of learning the specifications of missing operators.

The learning algorithm described in the previous chapter 5, figure 6–1 involves a recursive procedure, for determining the preconditions for the next stage of the problem and for recording the contribution of each operator involved in the solution to the problem. The same applies for the revised learning presented in this chapter, figure 6–2. A full description of the Extended-EBL program can be found in appendix B.

The main procedure of the revised learning algorithm is presented below, *ie* `extended_precondition_analysis`.

1. Identify the preconditions and the branching structure for the current operator. Record empty specifications for unknown or unidentified operators.

2. Recurse through the rest of the structure of the problem, following the branching structure and return the preconditions for each of the branches of the next stages.

3. Find the preconditions of the missing operators and revise the relevant unknown operators to reflect these learned preconditions.

4. Apply the matching process between the effects of the current operator and the preconditions of the next stage. Those preconditions which match

```
extended_precondition_analysis(Step,Back_propagated_preconditions) :-
        operator_structure(Operator,Step,
             Preconditions,Effects_structure),
        revise_rest_of_plan(Effects_structure,
             Preconditions_for_next_stages),
        match_sets_of_effects_with_preconditions(
             Effects_structure,
             Preconditions_for_next_stages,
             Set_of_Matched_preconditions,
             Set_of_Unmatched_preconditions),
        find_back_propagated_preconditions(
             Operator,
             Set_of_Matched_preconditions,
             Set_of_Unmatched_preconditions,
             Preconditions,Effects_structure,
             Back_propagated_preconditions).
extended_precondition_analysis(Step,[]) :-
        not(operator_structure(_,Step,_,_)),!.

revise_rest_of_plan([],[]) :- !.
revise_rest_of_plan([effects(Step,_)|Rest_of_effects_structure],
             [Preconditions_for_next_stage|Rest_of_preconditions]) :-
        extended_precondition_analysis(Step,Preconditions_for_next_stage),
        revise_rest_of_plan(Rest_of_effects_structure,
             Rest_of_preconditions).
```

Figure 6–1: PROLOG code for the Extended Precondition Analysis procedure

```
extended_precondition_analysis(Step,Back_propagated_preconditions,
            Another_operator,Another_state) :-
        operator_structure(Operator,Step,
            Preconditions,Effects_structure),
        revise_rest_of_plan(Effects_structure,
            Preconds_for_next_stages,
            Ops_with_missing_preconds,
            States),
        revise_ops_with_missing_pre(
            Effects_structure,
            Ops_with_missing_preconds,
            States,
            Preconds_for_next_stages,
            Revised_preconds),
        match_sets_of_effects_with_preconditions(
            Revised_preconds,Step,
            Effects_structure,
            Revised_effects_structure,
            Postconditions,
            Preconditions_for_next_stages,
            Set_of_Matched_preconditions,
            Set_of_Unmatched_preconditions),
        revise_op_with_missing_post(
            Operator,Step,
            Preconditions,Postconditions,
            Another_operator,Another_state),
        find_back_propagated_preconditions(
            Operator,
            Set_of_Matched_preconditions,
            Set_of_Unmatched_preconditions,
            Preconditions,Revised_effects_structure,
            Back_propagated_preconditions).
extended_precondition_analysis(Step,[]) :-
        not(operator_structure(_,Step,_,_)),!.

revise_rest_of_plan([],[]) :- !.
revise_rest_of_plan([effects(Step,_)|Rest_of_effects_structure],
            [Preconds_for_next_stage|Rest_of_preconditions]) :-
        extended_precondition_analysis(Step,Preconds_for_next_stage,
            Another_operator,Another_state),
        revise_rest_of_plan(Rest_of_effects_structure,
            Rest_of_preconditions).
```

Figure 6–2: Revised PROLOG code for the Extended Precondition Analysis procedure

effects are added to the list of `matched preconditions`. Those which do not match are added to the list of `unmatched preconditions`. At a leaf node there are no matched or unmatched preconditions, since no more operators follow.

5. Revise the specifications of the unknown operator (if there is one at this level of the problem) by adding the newly learned postconditions or effects to the previously unidentified operator.

6. Propagate the unmatched preconditions back through the current operator and add the propagated preconditions to the preconditions of the previous operator in the sequence.

The procedure for recursing through the problem, `revise_rest_of_plan`, involves a call to the main procedure above, and is as follows:

1. Call on `extended_precondition_analysis` to return the preconditions for the next stage for the current branch.

2. Recursively call this whole procedure to determine the preconditions for the next stage for the remaining branches. The recursion is complete when a termination node is reached, *ie* at one of the leaves of the tree where a sub part of the problem has been solved. At the leaf node, there are no more operators to follow and, so, no preconditions for the next stage.

The following clauses of the main procedure

1. `match_sets_of_effects_with_preconditions`

2. `find_back_propagated_preconditions`

3. `revise_ops_with_missing_pre`

4. `revise_op_with_missing_post`

make great use of the meta-language. The first to supervise the inferences required for the matching process between the effects of one operator and the preconditions of another. The second to deal with the effects of the state transitions on the back propagated preconditions. The third and fourth to determine the preconditions and effects of missing or unknown operators. The next three sections explain more fully how the meta-language helps to guide these processes.

6.3 Extending the matching process

In LP, the specifications for operators are represented by preconditions and postconditions described at the meta-level. The preconditions refer to properties of the state of the problem to which the operator can be applied. The postconditions refer to the properties of the state of the problem which should exist after the operators have been applied.

The same meta-level terms provide a language for describing the state of the problem before and after the application of an operator, *viz* the preconditions and postconditions. The form of these terms are:

```
predicate(Arguments)
```

where `predicate` describes the property or relationship that occurs between a set of arguments, denoted by `Arguments`, which are themselves object-level descriptions of whole or part of the problem state. Thus, the specification of an operator is described *all* in terms of properties of the problem state. For the sake of the discussion of Extended-EBL in chapter 5, the same assumption was made.

However, the specifications for operators need not be restricted to descriptions of the representation of the state *alone*. Instead of describing the *postconditions* of an operators, the *effects* of the operators can be represented. The effects refer to the relationship between the state before and after the application of the operator. In other words, effects represent the *state transition* that results from

applying the operator, rather than the property of the new state, as described by the postconditions.

An example of the difference between postconditions and effects is now described in terms of the meta-theory taken from chapter 3. The preconditions and effects of the `parm_tac` proof operator, which strips away universal quantifiers, `QTerm`, from the formula, `Fm`, resulting in the new formula, `Form` are:

Preconditions

```
goal(Fm)
decompose(Fm,QTerm,Form)
universal(Set_of_vars,Fm)
contain(Form,Set_of_vars)
```

Effects

```
include(Set_of_vars,Hyplist1,Hyplist2)
remove_quantifiers(Fm,QTerm,Form)
```

These effects could be represented by the following postconditions:

```
hypothesis(Set_of_vars,Hyplist2)
goal(Form)
```

The form of these effects is similar to that for preconditions:

```
predicate(Arguments)
```

except that `predicate` now describes the state transitions that has occurred between the set of arguments, `Arguments`, which contains the states before and after the application of the operator.

Such specifications, preconditions and effects, capture knowledge about both the state of the problem to which the operator applies and also about the state transition that results from its application. In addition, the effects provide more knowledge about the contribution of the operator, by representing, more explicitly, the state transition caused by the operator.

Although, the form of the meta-level terms for describing the preconditions and effects is very similar, the content of these terms is very different. As a result, the matching process for Extended-EBL, described in chapter 5, needs to be extended, since matching preconditions with effects now requires more than just a simple unification of terms. Instead, an inference must be made between the two terms, preconditions and effects, before a match, and thus a dependency link, can be considered valid.

The aim of the matching process is to determine how the effects of one operator contribute to the plan by permitting the preconditions of another operator to be satisfied. Thus, this inference must reflect the fact that the precondition can be inferred from the effect, even though the meta-level terms that describe them do not unify.

The form of this inference is an *implication* such as:

```
other_conditions & effect → precondition
```

where `other_conditions` reflects other conditions that must be true for the implication to be valid.

For the example described above involving the `parm_tac` operator, the implication required between the effect,

```
include(Set_of_vars,Hyplist1,Hyplist2)
```

and the precondition of another operator, say `hypothesis(Var1,Hyplist2)`, is

```
member(Hyp,List_of_hyps)
include(List_of_hyps,Old_hyplist,New_hyplist) →
      hypothesis(Hyp,New_hyplist)
```

Further examples of these implications are provided in a table of implications in chapter 7.

The meta-language must contain a table of implications, such that for each of the possible effects of an operator, the table must contain implications for all the preconditions that *should* occur directly as a result of the effects.

Thus, the table of implications describes a list of possible implications between the effects of one operator and the preconditions of another. The idea is that for each effect of an operator at a particular part of the problem, a match is attempted with each of the preconditions for the next stage. The match is successful if an implication can be found that links an effect to a precondition. There may be more than one successful match. This occurs when the state transition, caused by the effect of the operator, contributes to more than one precondition. The preconditions may belong to the same operator or more than one operator.

If an effect does not match one of the preconditions of the next stage then this is because there is no implication in the table that links the two meta-level terms. This situation arises if the table of implications is not complete. This could be remedied by attempting to learn an implication which links the effect to one of the preconditions. Unfortunately, this is non-trivial, since the effect may contribute to one or more of the preconditions of the other operators involved in the next stages of the plan. Thus, the ability to learn such implications is not pursued within this thesis.

The matching process forms an important component of the whole learning approach described within Extended-EBL. The table of implications plays a prominent role in the matching process. Without it the dependency links cannot be formed between the specifications of each operator involved in the plan. Such dependency links are required to sort out the matched from the unmatched preconditions which are passed onto the back propagation process. They are also needed to determine the contribution of the operator in producing the matched preconditions which are required for other operators involved later on in the plan.

For some domains a successful match may involve a simple unification of terms; for others, an implication may be required. Representing the definition

of a sucessful match *explicitly* with the aid of a table of implications, allows Extended-EBL to cater for various domains.

6.4 Back propagating over state transitions

The simple back propagation process, described in chapter 5, involved appending the unmatched preconditions, which result from the matching process, to the existing preconditions for each operator. The rationale was that any of the preconditions of the next stage which did not match the effects of the current operator, *ie* unmatched preconditions, must have been produced by the effects of previous operators. Thus, the properties of the problem state represented by these preconditions must have been present before the application of the current operator.

Since the analysis in chapter 5 represented the specifications of operators as propositions, the unmatched preconditions were simply appended to the preconditions of the current operator. However, providing more flesh to the content of the specifications of the operators, means that the previous description of the back propagation process must be extended to take into account the effects of the state transitions on the back propagated preconditions.

6.4.1 The role of the effects of an operator

State transitions change the object-level descriptions of states which are used as arguments of the meta-level preconditions and effects. Back propagating preconditions over state transitions requires that some of the arguments of these preconditions must be replaced to reflect the state transitions that have occurred due to the operator. Without these changes the back propagated preconditions [2] would be inconsistent with the object-level description of the state of the prob-

[2]Previously referred to as unmatched preconditions

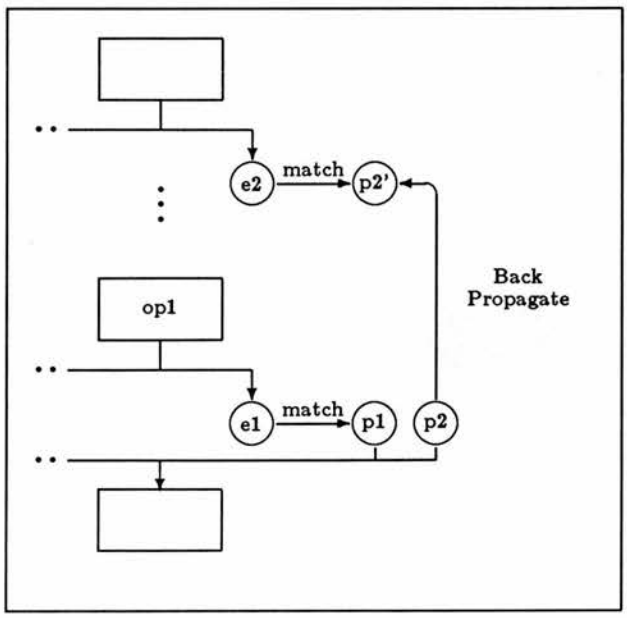


Figure 6–3: Resulting links due to the revised matching and back propagation processes

lem. The effects which describe these state transitions must be taken into account in the back propagation process so as to provide back propagated preconditions which are consistent with the previous object-level description of the problem state.

Figure 6–3 shows how dependency links are formed between the effects of one operator and the preconditions of another, *eg* the link between effect `e1` and precondition `p1`. However, for dependency links that span over one or more operators , as for effect `e2` and precondition `p2`, the back propagated precondition `p2'` must reflect the state transition caused by operator, `op1`.

The revised back propagation process takes into account the effects of each operator when propagating the unmatched preconditions back through the operator. The process may be complicated further by operators which generate more than one state of the problem to proceed with next. An example of this for solutions involving the process of goal reduction where an operator breaks up the problem into several sub problems each with sub goals to achieve. Each sub

goal has an associated set of effects. This means that many sets of unmatched preconditions may be propagated through the operators, where the number of sets depends on the number of sub goals.

For some operators, some of the resulting back propagated preconditions from one set of sub goals may be identical to some from another set of sub goals. Thus, these duplicate preconditions are removed. An example of this is shown in the chapter 7.

The back propagation process relies on the effects to provide a good description of the state transition caused by the operator. As a result, the back propagated preconditions provide the most general properties of the problem state to which the rest of the plan may be applied and may be used to match the effects of previous operators in the plan.

6.4.2 Object and meta-level interaction

Although, the effects provide a great deal of information about the state transitions caused by the operator, they do not provide sufficient information for the back propagation process to keep *all* of the back propagated preconditions in line with the object-level description of the state before the state transition. Thus, Extended-EBL must re-express the meta-level arguments of some of the back propagated preconditions in order to keep them consistent with the object-level description of the previous proof state.

Back Propagating the preconditions through the operator by reversing the changes produced by this effect, should result in the preconditions being consistent with the object-level description of the state prior to the state transition. However, this is not always the case. The problem is that the back propagation process only takes into account the direct effects of an operator on the unmatched preconditions. This is fine for unmatched preconditions that contain arguments explicitly referred to in the effects. But, for other preconditions which contain arguments that are not referred to directly by the effects, the back propaga-

tion process in its current form will not produce the most general or consistent preconditions for each level of the solution to the problem.

For example, assume that an operator has the following effect

```
replace_all(Old_Term,New_Term,Old_State,New_State)
```

which says that within Old_State, Old_Term is replaced by New_Term to produce New_State. Also, the next part of the problem comprises the preconditions

```
contain(New_State,New_Term)
decompose(New_State,Other_Term,Sub_State)
```

The decompose precondition says that New_State comprises a term, Other_Term and another state, Sub_State. Assume that this precondition is one of the unmatched preconditions because there is no implication that links it with any of the effects. When it is back propagated, replacing occurrences of New_Term by Old_Term and New_State by Old_State, the resulting precondition is

```
decompose(Old_State,Other_Term,Sub_State)
```

However, Sub_State could still contain occurrences of New_Term since it is part of New_State.

To complicate matters further, if the operator had two similar effects which generated two sub-goals to solve, *ie*

```
replace_all(Old_Term,New_Term1,Old_State,New_State1)
replace_all(Old_Term,New_Term2,Old_State,New_State2)
```

with sub parts described by

```
contain(New_State1,New_Term1)
decompose(New_State1,Other_Term,Sub_State1)
contain(New_State2,New_Term2)
decompose(New_State2,Other_Term,Sub_State2)
```

then the back propagated preconditions would include two distinct preconditions

```
decompose(Old_State,Other_Term,Sub_State1)
decompose(Old_State,Other_Term,Sub_State2)
```

both of which are not consistent for `Old_State`, since `Sub_State1` and `Sub_State2` are part of `New_State1` and `New_State2` respectively.

The solution to this problem involves making use of the available object-level knowledge to transform the meta-level terms such that they are fully defined in the language of the previous object-level state.

The procedure involves generating a term, in this case,

```
decompose(Old_State,Other_Term,X)
```

where `X` is a variable. By checking this term with the object-level state of the problem, `X`, is instantiated with the sub state, `Sub_State3`, which is part of `Old_State`. Effectively, this is like asking the question, "I know that the formula, `Old_State`, can be decomposed into a term, denoted by `Other_Term`, and a sub state, but I'm not sure what that other formula is, can you help?". The question can be easily answered by checking the object-level description of the formula, `Old_State` and returning the sub state, `Sub_State3`.

The problem of fully defining *all* the back propagated terms in the language of the object-level state before the application of the operator is solved by encouraging the interaction of both object and meta-level knowledge in order to check the validity of the meta-level terms and determine the necessary revisions, to bring them in line with the object-level state. As the result, the back propagation process is strengthened further.

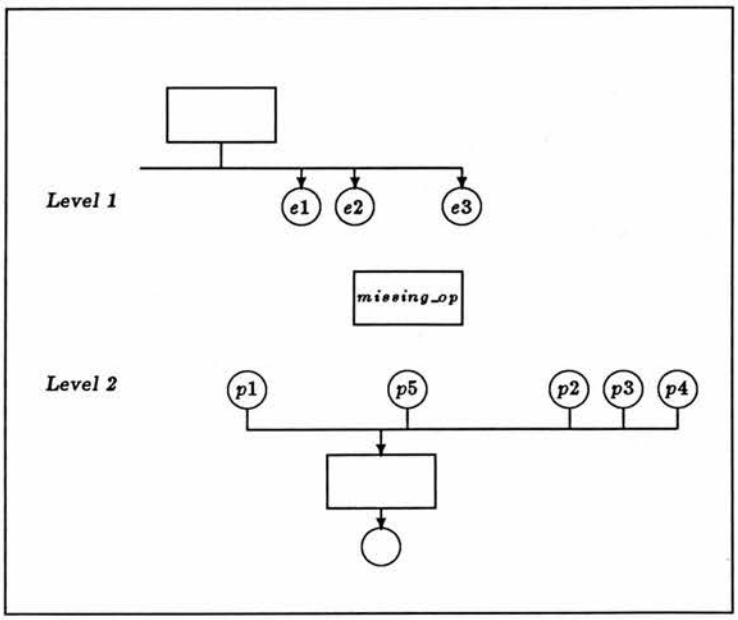


Figure 6–4: Representation of the known operators

6.5 Learning specifications of missing operators

This section fulfills a promise, made back in chapter 5, for the Extended-EBL program to learn the specifications of missing operators. The description of the original Extended-EBL technique did not permit this facility, because of the lack of detail about the form and content of the control knowledge learned. However, the analysis within this chapter, relating to the interaction between the object and meta-level knowledge and the generation of dependency links from the table of implications, provides the necessary mechanisms for learning the specifications of missing operators.

Assume that Extended-EBL has no knowledge of a particular operator involved in the solution to the problem. The situation after identifying all *known* operators is as shown in figure 6–4.

Although, Extended-EBL does not know any of the specifications of the missing operator, it does have knowledge, from the solution about the changes caused

by the unknown operator at the object-level, and the meta-level preconditions and effects of the other known operators involved in the proof, together with possible implications between effects and preconditions provided by the table of implications. In particular, it knows the preconditions for the next stage of the plan, in this case, $p1 - p5$, at level 2 and the effects of the previous operator, $e1 - e3$, at level 1.

6.5.1 Learning the effects of missing operators

The effects of the missing operator can be found by first identifying those preconditions for the next stage of the plan that could not possibly have existed in the state before the missing operator. The missing effects can then be determined by a process of *abduction* [Kowalski 79b] between those preconditions and the table of implications. By matching the *rhs* of one of the implications with one of those preconditions, the *lhs* of that implication denotes the missing effect.

The rationale behind this approach is that since the table of implications signifies possible implications between effects and preconditions, then, given these preconditions, one (or more) implications help in determining the missing effects.

The procedure for determining the missing effects is as follows:

- Check which of the meta-level preconditions for the next stage of the plan, eg $p1 - p5$, are implied by the object-level description of the state before the application of the missing operator, *ie* provided by the effects, $e1 - e3$. The resulting terms are the unmatched preconditions.

- Those preconditions which are not implied may be used to determine the effects of the missing operator. These effects are found by searching for implications which imply these preconditions in the table of implications, and then by *abduction* inferring the missing effects.

Assume that for the above example, the preconditions $p1$-$p4$ are implied by the object-level state at level 1, so they are added to the set of unmatched

preconditions. The precondition, $p5$, is not consistent with the previous object-level state and so is added to the set of matched preconditions and used to determine the effects of the missing operator.

For each precondition in the set of matched preconditions, the table of implications is searched for an implication whose *rhs* matches the precondition, $p5$. A successful match results in the *lhs* of this implication generating one of the missing effects.

For example, if there is an implication

$$e5 \rightarrow p5$$

then $e5$ is one of the effects of the missing operator.

6.5.2 Learning the preconditions of missing operators

The preconditions of the missing operator can be found by a similar procedure to the one above, involving the table of implications and the effects of the previous operator in the plan structure. This time the missing preconditions are determined by a process of *modus ponens* between these effects and each of the implications in the table. By matching the *lhs* for each of the implications with the effects, the *rhs* of the implication provides possible preconditions. The set of these preconditions is compared with the object-level state before the missing operator and those which are consistent represent the preconditions of the missing operator.

The rationale behind this approach is that since the table of implications signifies possible implications between effects and preconditions, then, given these effects of the operator before the missing one, the implications help in determining the missing preconditions.

The procedure for determining the missing preconditions is as follows:

- Taking each of the effects, of the operator prior to the missing operator, apply the *modus ponens* rule of inference with these effects and all the implications in the table.

- Check that the resulting preconditions are consistent with the object-level state of the proof before the application of the missing operator. If they are consistent, then these are the preconditions of the missing operator.

For the above example, there are three effects, $e1 - e3$. Thus, for each effect there should be an implication of the form

$$e1 \rightarrow p1'$$

respectively for $e2$ and $e3$, resulting in the preconditions, $p1'$, $p2'$ and $p3'$.

In general, this procedure should capture most of the possible preconditions of the missing operator. It tends to provide extra preconditions which are consistent with the relevant object-level state, but actually irrelevant.

However, it may also fail to identify all of the true preconditions, especially those not generated by the previous operator.

6.5.3 Limitations

Unfortunately, the complete specifications of the missing operator cannot be learned by the above procedures. The procedures for learning both the preconditions and effects rely on having a complete set of possible implications between effects and preconditions. Without the correct implication, there is no way of completing the chain of implication for the necessary preconditions and effects of the missing operators.

In addition, the existing procedures are not sufficient for learning *all* the preconditions of the missing operator. Take, for instance, the situation where some previous operator, not the one immediately prior to the missing operator, produces an effect which *should* provide a dependency link with one of the preconditions of the missing operator. The current procedure would not be able to

determine that this precondition should belong to the missing operator, because it relies only on the effects of the operator immediately prior to the missing operator, to guide the search for missing preconditions.

In order to solve this problem and determine these missing preconditions, the procedure would have to involve the effects of all the operators in the plan. The increase in the search space for possible missing preconditions could be increased further by having to deal with learning the specifications of many missing operators.

Such issues have not been examined in depth within this thesis, and remain as limitations.

6.6 Related Work

6.6.1 EBG/LEX2

The constraint back propagation technique in EBG and LEX2 applies only to equations described at the object-level. The process of back propagating the weakest preconditions involves identifying and replacing terms in these equations so as to reflect the state transition that has occurred as a result of the application of the operator. Unfortunately, this can often be quite difficult if the preconditions of the operator are not well defined.

In Extended-EBL because the preconditions and effects are explicitly defined in meta-level terms, the process of back propagating these back through state transitions is much more straightforward. Because of the explicit representation of the state transition in the effects of operators this process is more easily describable.

In addition, the meta-level preconditions represent more general properties about the state of the problem than the object-level descriptors and, indeed, can capture propoerties that are not explicit in the object-level.

6.6.2 LP/Precondition Analysis

The ability to learn the specifications of missing operators in LP was brought into question in chapter 5. The reliance on information about the specifications of operator immediately before and after the missing operator was not found to be sufficient for generating accurate preconditions and effects.

Extended-EBL is more successful than LP in identifying the preconditions and effects of missing operators. This is because it makes use of object-level as well as meta-level knowledge to check whether possible preconditions and effects could have occurred in the states before and after the missing operator. Unfortunately, Extended-EBL is still not able to look beyond the specifications of operators immediately surrounding the missing operator. As a result, the learnt specifications of the missing operator cannot be guaranteed to be complete.

6.6.3 PET

Porter and Kibler's PET program [Porter 84] does make use of an explicit representation of the state transition caused by the operators, which they call a *relational model.* Their use of the relational model as an explicit representation of the operator transformation, mirrors the role of the effects in the Extended EBL approach, where such effects are meant to represent the state transitions produced by the operator, *viz*, the relationship between the state before the operator and the state after.

Both the relational model in PET and the effects in my work form an important part of the back propagation process. In the case of the PET program, the relational model provides a means of propagating constraints from the state after the operator to the state before. In the same way, the effects provide valuable information for back propagating preconditions through the state transition.

However, in PET, the relational models are determined by the learning system itself. The process for learning these relational models is rather *ad hoc*, as described previously in chapter 2, such that the validity of the back propagated

states is questionable. This is not the case in the Extended EBL approach, where the effects of each operator are already known [3]. Instead, emphasis is placed on learning how the effects are employed in the solutions to the problems. Furthermore, the interaction between the object and meta-level knowledge provides additional security about the validity of the back propagated terms.

6.6.4 GENESIS

The work of Mooney & DeJong, culminating in the GENESIS program, involves learning concepts for narrative understanding [DeJong 83,Mooney 85,DeJong 86]. GENESIS learns concepts comprising a network of known schemata, which have been identified from input narratives.

GENESIS addresses the issue of determining the dependency links between components, *ie* schemata, of the learned concept. The schemata in the network, comprise either *state* schemata, which describe facts about the narrative, such as static situations, or, *action* schemata, which describe dynamic events involved in the narrative. Slots in the action schemata provides links to state schemata. The state schemata provide the means for connecting action schemata within the learnt concept, by describing the effects of one action and the preconditions of another.

In the case, of the Extended EBL approach, the dependency links are provided by implications at the meta-level, joining the effects of one operator with the preconditions of another. In addition, the interaction between the object and meta-level knowledge provide a means of checking the validity of the implications, even if they span several operators.

Mooney & DeJong do not provide much detail about the construction the network of schemata, and, thus, do not discuss, in any great depth, the mechanism for generating the dependency links Nor do they mention any problems

---

[3]Learning the effects of an operator is a non-trivial process, so these are provided to the Extended-EBL.

about checking the validity of the implications between schemata. Instead, emphasis is placed on how these links may be used in the refinement of the learnt concepts.

6.7 Conclusions

This chapter provides much more depth to the analysis of Extended-EBL and describes extensions which go beyond previous efforts in EBL research. Chapter 5 showed how existing mechanisms within EBL could be integrated within a single program, Extended-EBL. The emphasis was on how the different types of control knowledge could be learned and combined with the structure of the operators to form a generalised plan. However, it did not provide the details of the form and content of the control knowledge learned. The analysis within this chapter, provides more flesh to the plans learned with Extended-EBL.

The following contributions are made within this chapter:

- The role of meta-knowledge in describing control knowledge has been shown to be vital for improving both the expressibility and generality of the learned proof plans.

- It has also been shown how the meta-language plays an important role in enhancing the matching and back propagation processes.

- The interaction between object and meta-level knowledge provides a means of keeping the generalised plan in line with the detailed object-level description of the proof steps.

- This interaction has been found to be useful for learning the specifications of missing operators.

The meta-theory plays a vital role not only in providing general descriptions of the specifications of operators but also for enhancing some of the learning

processes within Extended-EBL, *eg* for enhancing the back propagation technique and for determining the specifications of missing operators. However, the development of a meta-theory for any domain is not a trivial task. Effort is required is identifying not only the possible meta-level terms in the meta-theory, but also the possible operators and their specifications and, as has been shown within Extended-EBL, the table of implications. The latter is essential for determining the contributions of operators within the plan, represented by dependency links between the preconditions and effects.

The problem with relying on a meta-theory is trying to keep it up to date. Ideally, one would wish the meta-theory to be learned by the system as new solutions to problems are generated. This can be partially done within Extended-EBL. The specifications of unknown, and thus new, operators are added to the existing list of operators. The back propagated preconditions at each level of the plan provide meta-level terms which describe new complex properties of problems states to which the plan can be applied. However, the ability to learn new primitive meta-level terms is beyond the capabilities of Extended-EBL and most learning systems [4].

Nevertheless, the use of the meta-theory provides plans that are more generally applicable than those described at the object-level. Consider the `unfold` proof operation from the isolate proof presented back in chapter 3. An object-level operator could not describe this operation – there would have to be one object-level operator for each unfold rewrite rule, rather than one operator capable of applying any rule from an unbounded set of rules. Meta-level descriptions are required to capture the defining characteristics of this set and describe how each rule is to be used [5]. In addition, experience with LP, described within

---

[4]Recent work by [Muggleton 88] on the *inverse resolution* technique has shown that new predicates in first-order logic can now be learnt.

[5]This will be shown in more detail in the next chapter, where the isolate proof plan is analysed more fully.

chapters 2 and 4 has shown the power of the meta-theory for generalising the applicability of successful solutions to algebraic problems.

Extended-EBL provides the means of learning plans from examples. It shows hows the back propagation technique incorporated in both LEX2 and EBG may be lifted to the meta-level. It also generalises the precondition analysis technique in LP to cope with more general plan structures, *eg* trees, etc.

Summary

In this chapter, the following has been presented and discussed.

- The learning algorithm within Extended-EBL is reviewed and the role of the meta-language within the algorithm identified and discussed.
- The simple matching process described in chapter 5 is extended to deal with the generation of dependency links between effects and preconditions, as well as postconditions and preconditions.
- The back propagation process is made more explicit by using the effects of an operator to determine the necessary changes to the meta-level terms that are to be propagated back over the state transitions caused by the operator. In addition, the interaction between object and meta-level knowledge helps to maintain a consistent description for all the back propagated preconditions.
- The problem of learning the specifications of missing operators is reviewed and a solution described, which involves the interaction between object and meta-level knowledge.
- This approach has been contrasted with related work in the area of EBL.
- Conclusions are presented about the relevance of this work to mainstream machine learning research and some further work is briefly discussed.

Chapter 7

Application of Extended-EBL to Learning Proof Plans

7.1 Introduction

The previous part of the thesis has dealt with the representation and learning of control knowledge from a fairly abstract viewpoint. Extended-EBL has been described, a program which combines the best features of prior EBL research into learning control knowledge. The analysis of Extended-EBL has emphasised how past EBL techniques can be integrated within a single program. The role of a meta-language has been shown to be important for representing general yet expressive control knowledge and for enhancing some of the learning processes within Extended-EBL.

In chapter 3, two example proofs were described and the corresponding proof plans required to generate these proofs were briefly discussed. This chapter discusses the application of these two proofs to the program, Extended-EBL, in order to test how well it is able to learn these proof plans. The proof plans described within this chapter are those which are actually generated by the Extended-EBL program.

The outline of the chapter is as follows.

Section 7.2 describes the application of Extended-EBL to the insert proof. The role of meta-language within the program is discussed in the context of this example. It also shows how Extended-EBL may be used to learn the specification of missing operators, although with qualified success.

Section 7.3 describes the application of the learning program to another proof from which another proof plan is extracted, the ISOLATE proof plan of chapter 3.

Section 7.4 analyses the benefits and limitations of Extended-EBL within the domain of proof plans.

Finally, section 7.5 presents some conclusions about Extended-EBL, discussing the contribution it makes to the area of explanation-based learning.

7.2 Applying Extended-EBL to the insert proof

The proof of the insert predicate, $I(a, x, z)$, described in the chapter 3, shows the power of the NuPRL environment for generating existence proofs for formulae of the form:

$$\forall Inputs.\exists Output.\ Spec(Inputs, Output)$$

and for extracting an algorithm, alg, such that

$$\forall Inputs.\ Spec(Inputs, alg(Inputs))$$

This section discusses the application of Extended-EBL to this proof. For the sake of clarity, only a portion of the proof is applied to the program such that only a partial proof plan is described [1]. However, it is sufficient to discuss the benefits and limitations of the program for learning the complete proof plan.

---

[1]The full proof is described in appendix A and comprises a large network of proof operators. Thus, the number of meta-level terms associated with each part of the proof plan is quite large.

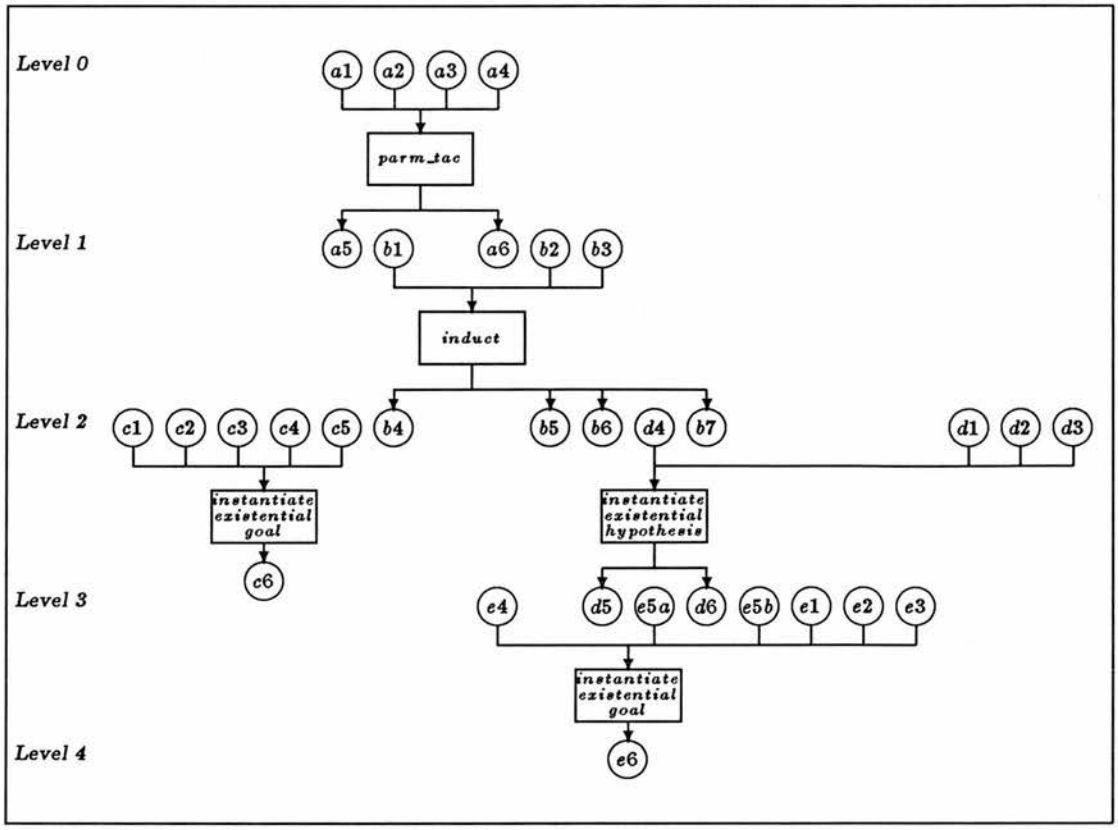


Figure 7–1: Representation of the proof operators

7.2.1 Learning about the insert proof

Figure 7–1 shows the structure, in this case a tree, of the proof operators involved in the proof. Table 7–1 presents the meta-level terms associated with each part of the proof and the resulting proof plan. Although, the structure of the operators is known, as a result of the operator identification phase [2], the contribution of each operator for each level of the proof is not known. This contribution is best represented by dependency links between the effects of one operator and the preconditions of another. Thus, one of the objectives of the learning algorithm is to identify these links.

---

[2]For the sake of simplicity, the operator identification stage involves simply matching the specifications of known operators with parts of the proof already described in meta-level terms.

Nodes	*Preconditions and Effects*
$a1$	`contain(fm2,[var1,var2])`
$a2$	`universal([var1,var2],fm)`
$a3$	`decompose(fm,qterm,fm2)`
$a4$	`goal(fm)`
$a5$	`include([var1,var2],hyplist1,hyplist2)`
$a6$	`remove_quantifier(fm,qterm,fm2)`
$b1$	`hypothesis(var2,hyplist2)`
$b2$	`goal(fm2)`
$b3$	`contain(fm2,[var2])`
$b4$	`replace_all(var2,nil,fm2,bfm1)`
$b5$	`replace_all(var2,[h1|t1],fm2,sfm1)`
$b6$	`replace_all(var2,t1,fm2,ifm1)`
$b7$	`include([h1,t1,ifm1],hyplist2,hyplist3)`
$c1$	`decompose(bfm1,qterm2,bfm2)`
$c1'$	`decompose(fm2,qterm2,fm3)`
$c2$	`existential([var3],bfm1)`
$c2'$	`existential([var3],fm2)`
$c3$	`contain(bfm2,[var3])`
$c3'$	`contain(fm3,[var3])`
$c4$	`hypothesis(var1,hyplist2)`
	`hypothesis(nil,hyplist2)`
$c4'$	`hypothesis(var1,hyplist2)`
	`hypothesis(var2,hyplist2)`
$c5$	`goal(bfm1)`
$c6$	`replace_all(var3,[var1|nil],bfm2,bfm3)`
$d1$	`decompose(ifm1,qterm2,ifm2)`
$d1'$	`decompose(fm2,qterm2,fm3)`
$d2$	`existential([var3],ifm1)`
$d2'$	`existential([var3],fm2)`
$d3$	`contain(ifm2,[var3])`
$d3'$	`contain(fm3,[var3])`
$d4$	`hypothesis(ifm1,hyplist3)`
$d5$	`replace_all(var3,t3,ifm2,ifm3)`
$d6$	`include([ifm3,t3],hyplist3,hyplist4)`
$e1$	`decompose(sfm1,qterm2,sfm2)`
$e1'$	`decompose(fm2,qterm2,fm3)`
$e2$	`existential([var3],sfm1)`
$e2'$	`existential([var3],fm2)`
$e3$	`contain(sfm2,[var3])`
$e3'$	`contain(fm3,[var3])`
$e4$	`goal(sfm1)`
$e5a$	`hypothesis(t3,hyplist4)`
$e5b$	`hypothesis(h1,hyplist4)`
$e5'$	`hypothesis(h1,hyplist3)`
$e6$	`replace_all(var3,[h1|t3],sfm2,sfm3)`

Applying the main procedure of the algorithm results in recursing through the proof tree until a leaf node is reached. Choosing the left most sub-tree at each branch of the tree, means that the first leaf node encountered, is the effect,

`replace_all(var3,[var1|nil],bfm2,bfm3)` denoted *c*6

Since there are no subsequent operators in this branch of the tree, the application of both the matching and back propagation processes is trivial, *ie* there are no matched or unmatched preconditions, so there are no back propagated preconditions.

The preconditions for the next stage of the plan at level 2, for this branch of the proof tree, are just the preconditions of the current operator, `instantiate_existential_goal`, *viz*

`decompose(bfm1,qterm2,bfm2)`	denoted *c*1
`existential([var3],bfm1)`	denoted *c*2
`contain(bfm2,[var3])`	denoted *c*3
`hypothesis(var1,hyplist2)` &	
`hypothesis(nil,hyplist2)`	denoted *c*4
`goal(bfm1)`	denoted *c*5

Thus, the base case of the insert proof comprises the formula, `bfm1`, which includes the existentially quantified variable, `var3`. Also `var1` and `nil` [3] must be part of the hypothesis list, `hyplist2`, since they are used to instantiate the existentially quantified variable, `var3`.

The remaining branch of the proof at level 2, the step case, is now dealt with. Since there are no subsequent operators at level 4, the application of the matching and back propagation processes is trivial.

However, at level 3 the matching process links the precondition

---

[3]Note that `nil` will be transformed to `var2` when the precondition, `hypothesis(nil,hyplist2)` is back propagated through the `induct` operator.

`hypothesis(t3,hyplist4)` denoted $e5a$

with the effect,

`include([ifm3,t3],hyplist3,hyplist4)` denoted $d6$

by the relevant implication 1, from table 7–2.

The table of implications for the effects involved in this proof is shown in table 7–2. They all involve implications between effects and possible preconditions. For each effect, the list of implications in table 7–2 is checked for one which links it to one or more of the preconditions. The effect $d5$ fails to match any of the preconditions. The only relevant implications would require the preconditions, `goal(ifm3)`, `contain(ifm3,t3)` or `hypothesis(ifm3,hyplist4)` to be present at level 3 of the plan. For the purposes of this discussion, they are not present [4]. As a result, $d5$ is now considered an effect of the overall proof plan.

Returning to the dependency link between $d6$ and $e5a$, the meaning of the implication 1 is as follows: since an operator has included a new hypothesis in the current hypothesis list, then, if this effect does contribute to the proof plan there must be a precondition mentioning this hypothesis. As a result, this precondition is added to the list of matched preconditions for this part of the proof.

The remaining unmatched preconditions are propagated back through the current operator. This involves taking into account the effects of the operator, `instantiate_existential_hypothesis`, *ie* $d5$ and $d6$, to replace some of the arguments of the back propagated terms, and also checking the rest of these arguments for consistency with the object-level description of the state of the proof before the application of the operator.

---

[4]This is not strictly true, since the hypothesis, `ifm3`, actually represents the inductive hypothesis which is required to terminate the step case of the proof. However, this proof has been drastically simplified for the sake of brevity.

Number	*Implications*
1	member(Hyp,List_of_hyps) include(List_of_hyps,Old_hyplist,New_hyplist) → hypothesis(Hyp,New_hyplist)
2	goal(Old_state) & base(Old_term,New_term,Ind_scheme) & replace_all(Old_term,New_term,Old_state,New_state) → goal(New_state)
3	goal(Old_state) & step(Old_term,New_term,Ind_scheme) & replace_all(Old_term,New_term,Old_state,New_state) → goal(New_state)
4	goal(Old_state) & ind_hypothesis(Old_term,New_term,Ind_scheme) & replace_all(Old_term,New_term,Old_state,New_state) → hypothesis(New_state,Hyp_list)
5	contain(Old_state,Old_term) & replace_all(Old_term,New_term,Old_state,New_state) → contain(New_state,New_term)
5a	exp_at(Old_state,Position,Old_term) & replace_all(Old_term,New_term,Old_state,New_state) → exp_at(New_state,Position,New_term)
6	goal(Old_state) & remove_quantifier(Old_state,Quantifiers,New_state) → goal(New_state)
7	goal(Old_state) & prim_rec(Exp,Arg) & rewrite(Position,step(Exp),Old_state,New_state) → goal(New_state)

Table 7–2: List of implications for effects involved in example proof

For the operator, `instantiate_existential_hypothesis`, back propagating the unmatched preconditions means reversing the state transitions described by the effects,

`replace_all(var3,t3,ifm2,ifm3)`	denoted $d5$
`include([ifm3,t3],hyplist3,hyplist4)`	denoted $d6$

Reversing the state transition caused by the first effect, $d5$, involves replacing any occurrences of the formula, `ifm3`, by `ifm2`, and the value `t3` by the term `var3`. For the second effect, $d6$, any occurrences of the hypothesis list, `hyplist4` are replaced by the hypothesis list, `hyplist3`, which existed before the application of the current operator.

The resulting back propagated preconditions are the following:

`hypotheses(h1,hyplist3)`	denoted $e5'$
`goal(sfm1)`	denoted $e4$
`decompose(sfm1,qterm2,sfm2)`	denoted $e1$
`existential([var3],sfm1)`	denoted $e2$
`contain(sfm2,[var3])`	denoted $e3$

Note that $e5'$ contains the hypothesis list, `hyplist3`, from the previous proof state.

Figure 7–2 shows the resulting partial plan at level 2. The link, labelled m, refers to the dependency link between the effect, $d6$, and the precondition, $e5a$, provided by the implication 1 from the table of implications. The links, labelled bp, connect the preconditions to their back propagated versions. There are no redundant terms.

All the preconditions for the next stages at level 2 have been determined, resulting in two sets of preconditions for the two branches of the proof, the base and step cases. The matching process may now be applied for each case.

For the base case, the effect,

`replace_all(var2,nil,fm2,bfm1)`	denoted $b4$

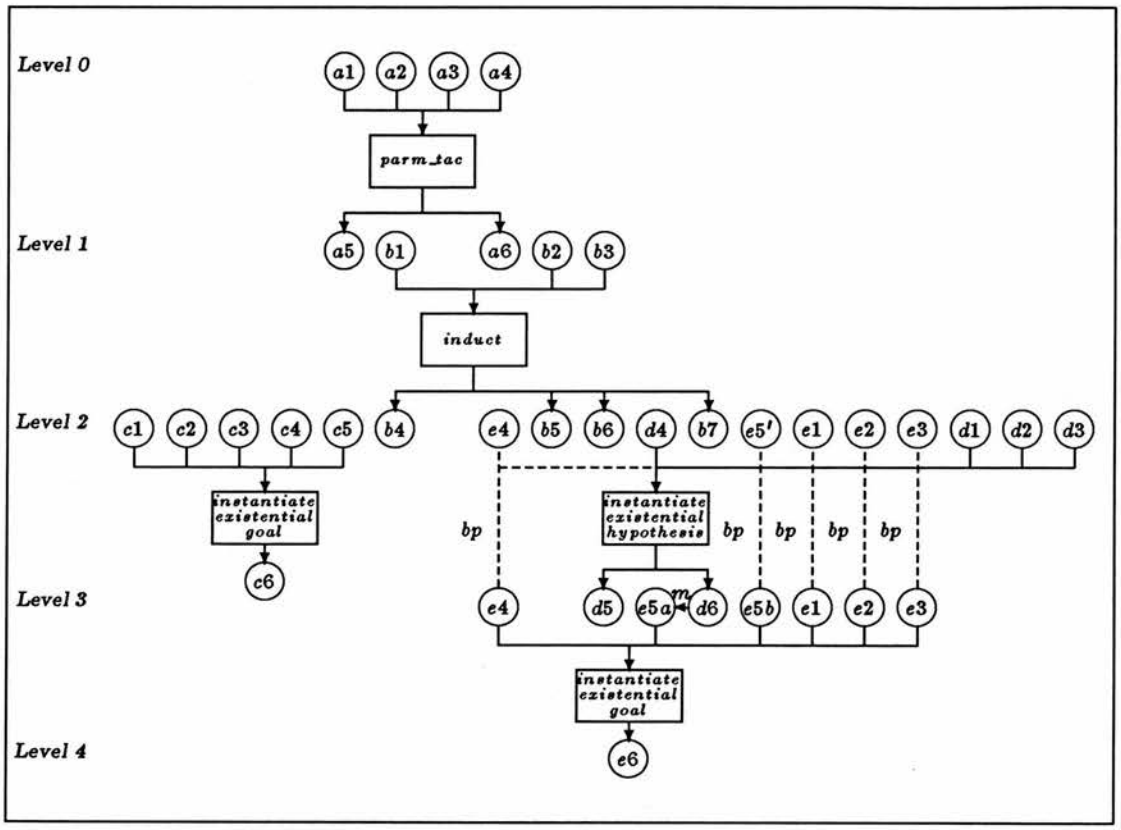


Figure 7–2: Representation of the partial plan after level 2

matches with the precondition, goal(bfm1), denoted by $c5$, as a result of implication 2 from table 7–2, *ie*

```
goal(Old_state) &
base(Old_term,New_term,Ind_scheme) &
replace_all(Old_term,New_term,Old_state,New_state)
      → goal(New_state)
```

The meaning of this implication is that if all occurrences of some term, denoted by Old_term, is replaced by another, New_term, in the formula, Old_state, resulting in the new formula, New_state, and if Old_term and New_term represent the inductive variable and the base value respectively of an inductive scheme represented by Ind_scheme, then the formula, New_state, which represents the base case of an inductive proof, is the next goal to be proved. Similar implication applies for the step case of an inductive proof, *ie* implications 3 and 4.

Note that there are other possible implications that involve the effect, replace_all other than for the base and step case of an inductive proof, which infer new goals to be proved. These reflect that other preconditions can be inferred from this effect.

The precondition, contain(New_state,New_term), may be inferred from the replace_all effect provided Old_state did indeed contain Old_term, denoted by contain(Old_state,Old_term), as repesented in implication 5. A variation on this is implication 5a which explicitly states the position of the terms in the both states. This latter infernce is not used in the current proof but in the following proof in section 7.3.

The remaining preconditions, for the base case branch, are added to the list of unmatched preconditions, *viz*

```
decompose(bfm1,qterm2,bfm2)                    denoted c1
existential([var3],bfm1)                       denoted c2
contain(bfm2,[var3])                           denoted c3
hypothesis(var1,hyplist2) &
```

Label/Effects	*Label/Preconditions*	*Imp*
($b5$) $replace\_all(var2, [h1\|t1], fm2, sfm1)$	($e4$) $goal(sfm1)$	3
($b7$) $include([h1, t1, ifm1],$	($e5'$) $hypothesis(h1, hyplist3)$	1
$hyplist2, hyplist3)$	($d4$) $hypothesis(ifm1, hyplist3)$	1
($b6$) $replace\_all(var2, t1, fm2, ifm1)$	($d4$) $hypothesis(ifm1, hyplist3)$	4

Table 7–3: Dependency links for the step case at level 2

```
hypothesis(nil,hyplist2)                    denoted c4
```

For the step case, there are several dependency links between the effects and various preconditions. The effect $b5$ matches the precondition $e4$ by implication 3. The effect $b7$ matches the preconditions $d4$ and $e5'$ both by implication 1. The effect $b6$ matches the precondition $d4$ by the implication 4. These are shown in table 7–3

The remaining preconditions are added to the unmatched preconditions at level 1, *viz*

```
decompose(ifm1,qterm2,ifm2)                 denoted d1
existential([var3],ifm1)                    denoted d2
contain(ifm2,[var3])                        denoted d3
decompose(sfm1,qterm2,sfm2)                 denoted e1
existential([var3],sfm1)                    denoted e2
contain(sfm2,[var3])                        denoted e3
```

The back propagation process is applied to all the unmatched preconditions, from the base and step cases. This involves the effects of the operator, `induct`, *ie* $b4$, $b5$, $b6$ and $b7$.

The state transition through the base case branch of the operator is described by the effect,

```
replace_all(var2,nil,fm2,bfm1)              denoted b4
```

This involves replacing all occurrences of `var2` in the formula, `fm2`, with the value `nil`, resulting in the new formula, `bfm1`. This produces the following preconditions:

```
existential([var3],fm2)
decompose(fm2,qterm2,bfm2)
contain(bfm2,[var3])
```

which only partially defines these preconditions with respect to the object-level description of the state, $\exists z.I(a,x,z)$, denoted by `fm2`. The formula $I(a,nil,z)$, denoted by `bfm2`, is not a sub-formula of the formula `fm2`. Thus, the meta-level term, `decompose(fm2,qterm2,bfm2)`, which states that the formula, `fm2`, can be decomposed into a set of quantified terms, `qterm2`, and the formula, `bfm2`, is incorrect. Furthermore, the other meta-level term, `contain(bfm2,[var3])`, is irrelevant,since the formula `bfm2` plays no part at this stage of the proof.

So, Extended-EBL makes use of the available object-level knowledge to revised these meta-level terms, resulting in the following back propagated preconditions

`decompose(fm2,qterm2,fm3)`	denoted $c1'$
`contain(fm3,[var3])`	denoted $c3'$

where `fm3` represents the formula, $I(a,x,z)$.

Figure 7–3 shows the resulting back-propagated terms, together with the other matching links for the base case branch of the `induct` operator.

After the replacements of various arguments and the check for consistency with the object-level state before the application of the operator, the resulting back propagated preconditions at level 1 are:

`decompose(fm2,qterm2,fm3)`	denoted $c1'$, $d1'$, $e1'$
`existential([var3],fm2)`	denoted $c2'$, $d2'$, $e2'$
`contain(fm3,[var3])`	denoted $c3'$, $d3'$, $e3'$
`hypothesis(var1,hyplist2)` &	
`hypothesis(var2,hyplist2)`	denoted $c4'$

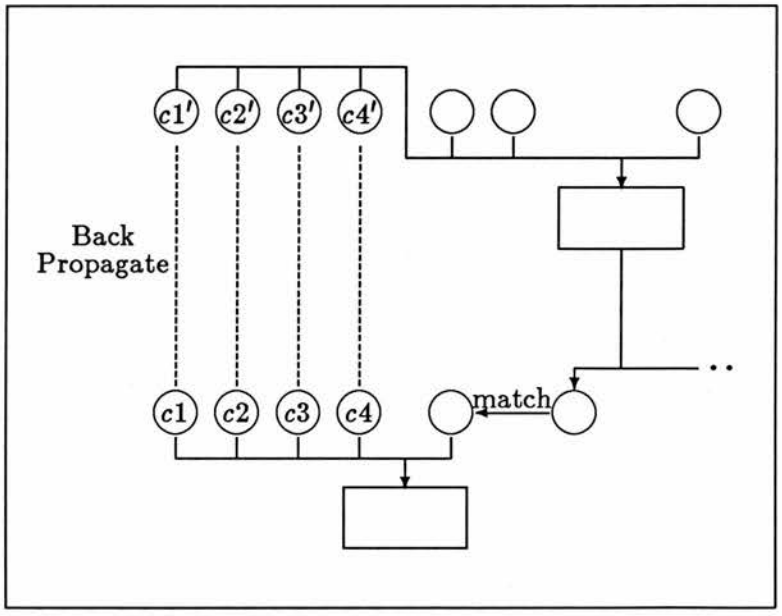


Figure 7–3: Links and back-propagated preconditions

Label/Effects	*Label/Preconditions*	*Impl*
(*a*6) *remove_quantifier*(*fm*, *qterm*, *fm*2)	(*b*2) *goal*(*fm*2)	6
(*a*5) *include*([*var*1, *var*2], *hyplist*1, *hyplist*2)	(*b*1) *hypothesis*(*var*2, *hyplist*2)	1
	(*c*4′) *hypothesis*(*var*1, *hyplist*2)	1

Table 7–4: Dependency links between effects and preconditions at level 1

There are multiple copies of the first three terms. The redundant copies are removed, leaving $c1'$, $c2'$ and $c3'$. Figure 7–4 shows the representation of the partial plan after level 1 of the proof tree, together with the relevant dependency links and back propagated terms.

The matching process is now applied at level 1, resulting in the following matches show in table 7–4 [5]. The remaining unmatched preconditions are prop-

[5]Note that the hypothesis var2 is required for more than one stage of the proof, since it has already been propagated back from a later stage of the plan.

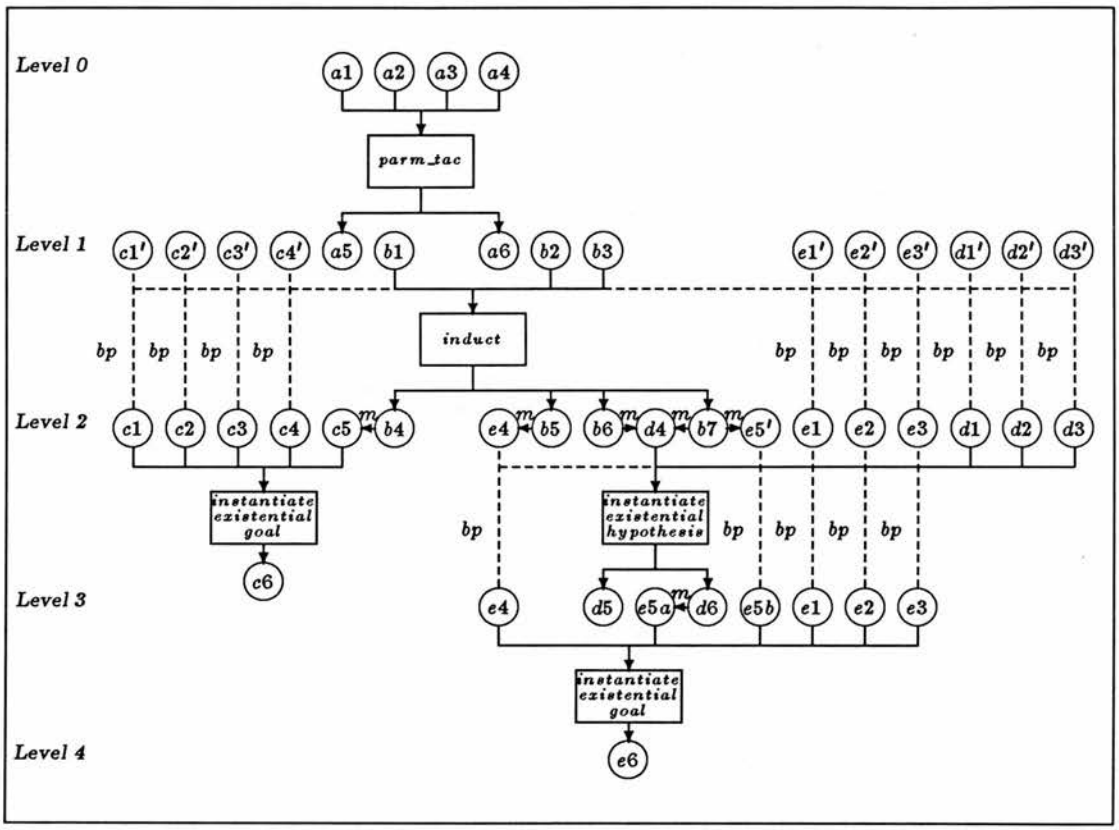


Figure 7–4: Representation of the partial plan after level 1

agated back through the operator, resulting in the following back propagated preconditions [6]

`decompose(fm2,qterm2,fm3)`	denoted $c1'$
`existential([var3],fm2)`	denoted $c2'$
`contain(fm3,[var3])`	denoted $c3'$
`contain(fm2,[var2])`	denoted $b3$

There are no copies of the same term. However, the term, `contain(fm2,[var2])` is redundant because it is subsumed by the term, `contain(fm2,[var1,var2])`, denoted by $a1$. Figure 7–5 shows the representation of the completed partial plan at level 0 of the proof.

7.2.2 The Learnt Proof Plan

Extended-EBL generates a proof plan incorporating the three types of control knowledge, discussed in previous chapters

- The applicability not only of each operator within the plan, but also of the plan itself at *many* levels of the proof. Thus, the preconditions at level 0 of the plan, $[a1, a2, a3, a4, c1', c2', c3']$, represent those properties which must exist in a formula for the entire plan to be applied. The preconditions at levels 1, 2 and 3 represent the applicability of the sub-plans within each branch of the proof. At level 2, there are two branches of the proof plan and thus sub-plans are required to prove both the base and step cases of the inductive proof.

- The contribution of each operator within the plan is provided by the effects of the operators, together with the relevant dependency links to the preconditions of the operators to which they contribute; *eg* for the operator,

---

[6]Note that the preconditions, $d1'$, $d2'$, $d3'$, $e1'$, $e2'$ and $e3'$ are redundant copies of the preconditions, $c1'$, $c2'$ and $c3'$.

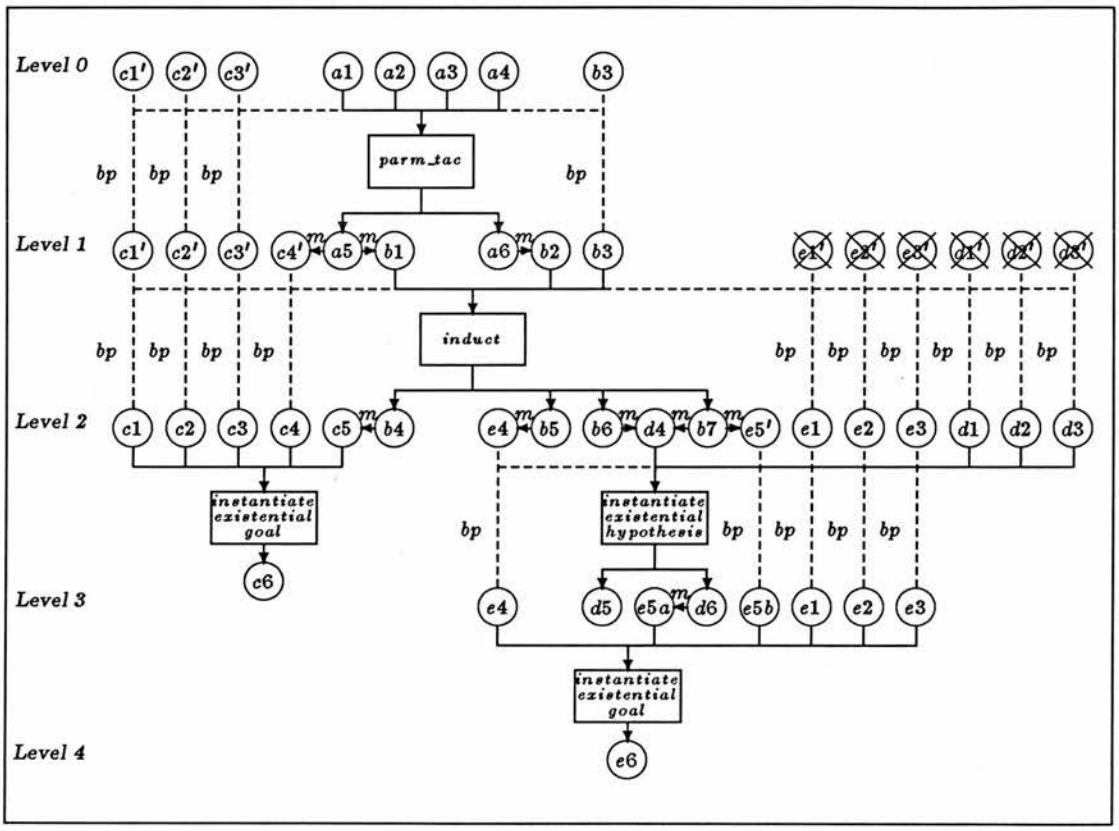


Figure 7–5: Representation of the completed partial plan

parm_tac, the effect $a5$ has a dependency link to the precondition, $c4$, of operator, instantiate_existential_goal, via $c4'$, and another link to the precondition, $b1$, of operator, induct. The other effect, $a6$, has a dependency link to the precondition, $b2$, also of the operator, induct. Together, these represent the contributions of the operator, parm_tac. In addition, the preconditions, $c1', c2', c3'$, link via back propagation links to the preconditions of the operator, instantiate_existential_goal at level 2.

- The tree structure of the proof is reflected in the structure of the proof plan shown in figure 7–5.

The preconditions, $c1', c2', c3'$, provide a restriction on the formula, $\exists z.I(a, x, z)$, denoted by fm2. They state that the overall proof plan may only be applied to formulae which contain existential quantifiers.

If Extended-EBL comprised only the original precondition analysis technique, then it would have been unable to identify such restrictions on the formula. In this case, the resulting proof plan could have been applied to formulae not involving existential quantifiers, since the preconditions for this plan would comprise just the preconditions of the first operator, parm_tac.

Appendix C shows the output obtained from the Extended-EBL program, when the full proof is applied. The proof plan described in figure 7–5 represents part of that required to prove the existence of the insert predicate, $I(a, x, z)$. The complete proof plan learned from this proof provides much more knowledge about the applicability of the entire network of proof operators such that it would be suitable for use in other existence proofs.

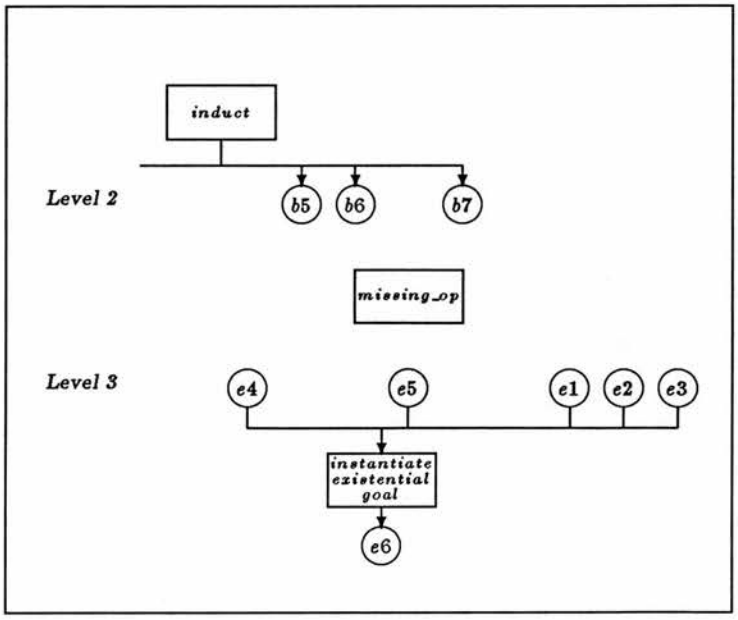


Figure 7–6: Representation of the proof operators

7.2.3 Learning specifications of missing operators

Assume that Extended-EBL has no knowledge of the proof operator, `instantiate_existential_hypothesis`, involved in the proof. The situation after identifying all *known* operators is as shown in figure 7–6 [7].

Although, Extended-EBL does not know any of the specifications of the missing operator, it does have knowledge, from the proof, about the changes caused by the unknown operator at the object-level, and the meta-level preconditions and effects of the other known operators involved in the proof, together with possible implications between effects and preconditions provided by the table of implications. In particular, it knows the preconditions for the next stage of the plan, in this case, $e1 - e5$, at level 3 and the effects of the previous operator, $b5 - b7$, at level 2.

---

[7] In order to focus on the issue of learning the specifications of the missing operator, only the relevant part of the step case of the partial proof plan is represented.

The effects of the missing operator are found by first identifying those preconditions for the next stage of the plan that did not exist in the state before the missing operator. Then, by matching the *rhs* of one of the implications from table 7–2 with one of these preconditions, the *lhs* of that implication denotes the missing effect.

For the above example, the preconditions, *e*1-*e*4 and *e*5*b*, are implied by the object-level state at level 2, so these could not have been introduced by the missing operator. The precondition, `hypothesis(t3,hyplist4)`, is not implied by the previous object-level state and is used to determine the effects of the missing operator.

For each precondition in the set of matched preconditions, the table of implications is searched for an implication whose *rhs* matches the precondition. A successful match results in the *lhs* of this implication generating one of the missing effects.

In this case, `hypothesis(t3,hyplist4)` matches the *rhs* of the implication 1,

```
include(Hyp,Old_hyplist,New_hyplist)
→ hypothesis(Hyp,New_hyplist)
```

resulting in the effect, `include([t3],hyplist3,hyplist4)` [8].

The preconditions of the missing operator are found by a similar procedure to the one above, involving the table of implications and the effects of the previous operator in the plan structure. By matching the *lhs* for each of the implications with the effects, the *rhs* of the implication provides possible preconditions. The set of these preconditions is compared with the object-level state before the

---

[8]Note that only one effect is determined for the missing operator, whereas the true operator, `instantiate_existential_hypothesis`, has two effects. The other effect is not determined because only part of the proof is shown. The other effect provides information about the inductive hypothesis, which is actually used later on in the proof.

missing operator and those which are implied represent the preconditions of the missing operator.

For the effect, `replace_all(var2,[h1|t1],fm2,sfm1)`, denoted by *b*5, applying modus ponens with implications 2 and 3 from the table of implications, results in the preconditions, `goal(sfm1)` and `contain(sfm1,[h1|t1])`, which are both implied by the object-level state of the proof at level 2.

For the effect, `replace_all(var2,t1,fm2,ifm1)`, denoted by *b*6, applying the modus ponens rule with implication 4, results in the valid preconditions, `hypothesis(ifm1,hyplist3)` and `contain(ifm1,[t1])`.

For the remaining effect, `include([h1,t1,ifm1],hyplist2,hyplist3)`, denoted by *b*7, the resulting preconditions are

```
hypothesis(h1,hyplist3)
hypothesis(t1,hyplist3)
hypothesis(ifm1,hyplist3)
```

Thus, the preconditions of the missing operator are

```
goal(sfm1)
contain(sfm1,[h1|t1])
contain(ifm1,[t1])
hypothesis(h1,hyplist3)
hypothesis(t1,hyplist3)
hypothesis(ifm1,hyplist3)
```

In general, this procedure should capture most of the possible preconditions of the missing operator. It tends to provide extra preconditions which are consistent with the relevant object-level state, but actually irrelevant.

However, for this particular operator, `instantiate_existential_hypothesis`, it fails to identify some of the true preconditions

```
existential([var3],ifm1)
```

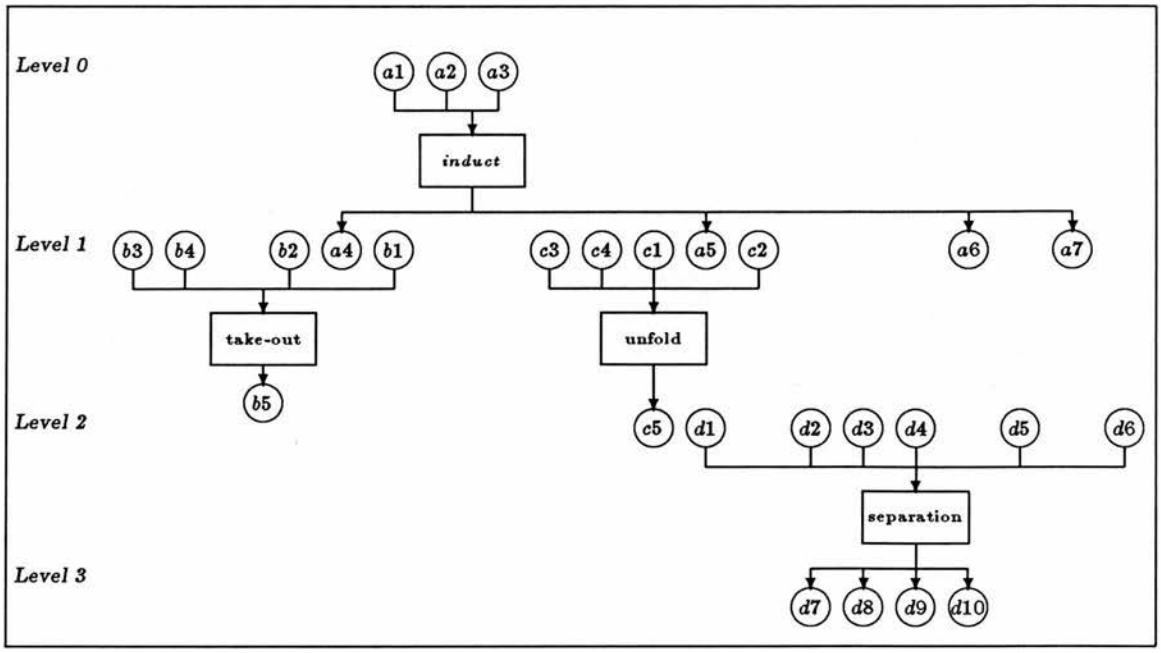


Figure 7–7: Representation of the IMPRESS plan after level 2

```
decompose(ifm1,qterm2,ifm2)
contain(ifm2,[var3])
```

This is because these preconditions were not generated by the previous operator, or, indeed, by any of the previous operators. Instead, these properties were present in the initial state of the proof, and so cannot be determined by the above procedure.

7.3 Learning the IMPRESS proof plan

The next example shows how Extended-EBL is able to learn a complete proof plan from the isolate proof described in chapter 3. This proof plan provides more knowledge about the role of the inductive hypothesis within the inductive proof.

Figure 7–7 shows the structure, in this case a tree, of the proof operators involved in the proof. Table 7–5 presents the meta-level terms associated with

Nodes	*Preconditions and Effects*
$a1$	`goal(fm1)`
$a2$	`hypothesis(var,hyplist1)`
$a3$	`contain(fm1,[var])`
$a4$	`replace_all(var,nil,fm1,bfm1)`
$a5$	`replace_all(var,[h1|t1],fm1,sfm1)`
$a6$	`include([h1,t1,ifm],hyplist1,hyplist2)`
$a7$	`replace_all(var,t1,fm1,ifm1)`
$b1$	`goal(bfm1)`
$b2$	`exp_at(bfm1,[n|posn],nil)`
$b3$	`exp_at(bfm1,[0|posn],bfm2)`
$b3'$	`exp_at(fm1,[0|posn],fm2)`
$b4$	`prim_rec(bfm2,n)`
$b4'$	`prim_rec(fm2,n)`
$b5$	`rewrite(posn,base(bfm2),bfm1,bfm3)`
$c1$	`goal(sfm1)`
$c2$	`exp_at(sfm1,[n|posn],[h1|t1])`
$c3$	`exp_at(sfm1,[0|posn],sfm2)`
$c3'$	`exp_at(fm1,[0|posn],fm2)`
$c4$	`prim_rec(sfm2,n)`
$c4'$	`prim_rec(fm2,n)`
$c5$	`rewrite(posn,step(sfm2),sfm1,sfm3)`
$d1$	`goal(sfm3)`
$d2$	`contain(sfm3,ants)`
$d2'$	`contain(fm1,ants)`
$d3$	`contain(sfm3,cons)`
$d3'$	`contain(fm1,cons)`
$d4$	`contain(sfm3,prec)`
$d4'$	`contain(fm1,prec)`
$d5$	`hypothesis(ifm,hyplist2)`
$d6$	`implication(sfm3,ants,cons,prec)`
$d6'$	`implication(fm1,ants,cons,prec)`
$d7$	`goal(sfm4)`
$d8$	`implication(sfm4,ants2,cons2,nil)`
$d9$	`notcontain(sfm4,ants2)`
$d10$	`notcontain(sfm4,cons2)`

Table 7–5: Preconditions and effects of the proof operators for the IMPRESS proof plan

each part of the proof and resulting proof plan. Although, the structure of the operators is known, as a result of the operator identification phase [9], the contribution of each operator for each level of the proof is not known. This contribution is best represented by dependency links between the effects of one operator and the preconditions of another. Thus, one of the objectives of the learning algorithm is to identify these links.

Applying the main procedure of the algorithm, results in recursing through the proof tree until a leaf node is reached. Choosing the left most sub-tree at each branch of the tree, means that the first leaf node encountered, is the effect,

`rewrite(posn,base(bfm2),bfm1,bfm3)` denoted *b*5

There are no subsequent operators in this branch of the tree, such that the preconditions for the base case of the proof at level 1 are the preconditions of the `take_out` operator, *viz*

`goal(bfm1)` denoted *b*1
`exp_at(bfm1,[n|posn],nil)` denoted *b*2
`exp_at(bfm1,[0|posn],bfm2)` denoted *b*3
`prim_rec(bfm2,n)` denoted *b*4

Before the program can proceed further back through the proof, it must propagate preconditions from the remaining branches of the proof at level 1, *ie* the step case branch of the proof tree. For the step case, there are no more operators beyond level 3, thus the matching process at level 2 links the precondition

`goal(sfm3)` denoted *d*1

with the effect,

---

[9]The assumption here is that the operator identification mechanism adopted within the LP program and described in chapter 4 is suitable for identifying the correct operators at each stage in the proof.

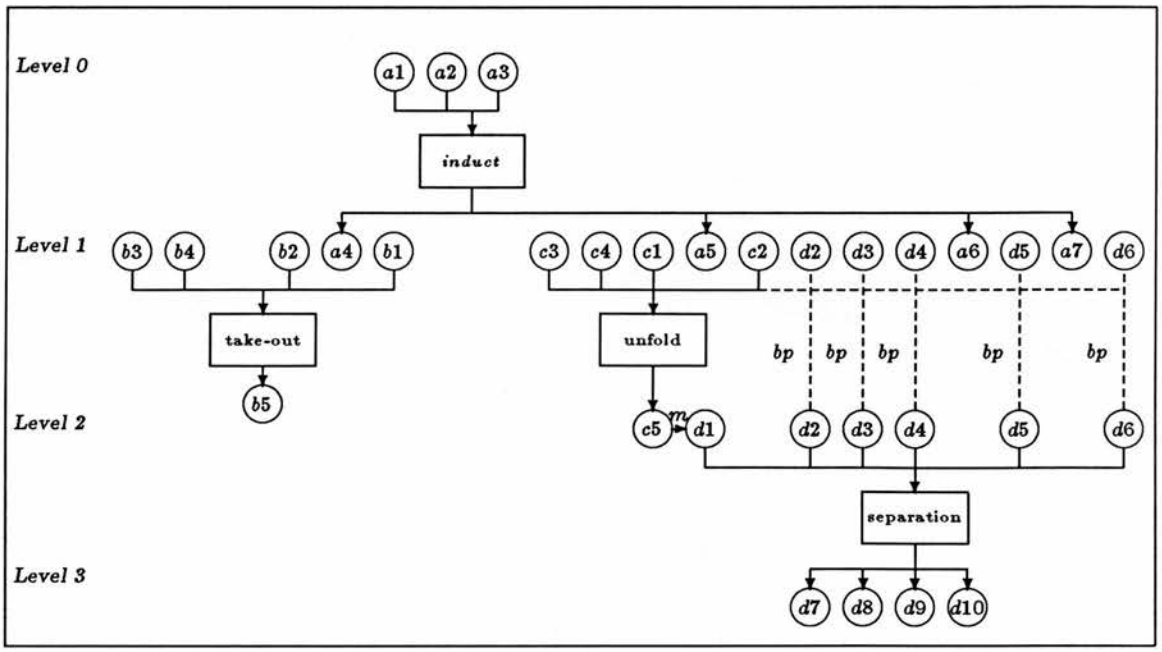


Figure 7–8: Representation of the IMPRESS plan after level 1

`rewrite(posn,step(sfm2),sfm1,sfm3)` denoted $c5$

by the relevant implication 7, from table 7–2. This precondition is added to the list of matched preconditions for this part of the proof.

The remaining preconditions are propagated back through the `unfold` operator taking into account the effects of the operator, *ie* $c5$. This involves replacing the occurrences of `sfm3` by `sfm1` and re-expressing any other meta-level arguments that are not dealt with directly by the effects.

The resulting back propagated preconditions are the following:

`contain(sfm1,Ants)`	denoted $d2$
`contain(sfm1,Cons)`	denoted $d3$
`contain(sfm1,Prec)`	denoted $d4$
`implication(ifm,Ants,Cons,Prec)`	denoted $d5$
`hypothesis(ifm,hyplist2)`	denoted $d6$

Figure 7–8 shows the resulting partial plan at level 1. The link, labelled m, refers to the dependency link between the effect, $c5$, and the precondition, $d1$, provided by the implication 5a [10] from the table of implications. The links, labelled bp, connect the preconditions to their back propagated versions. There are no redundant terms.

All the preconditions for the next stages from level 1 have been determined, resulting in two sets of preconditions for the two branches of the proof, the base and step cases. The matching process may now be applied for each case.

For the base case, the effect,

`replace_all(var,nil,fm,bfm1)` denoted $a4$

matches with the precondition, `goal(bfm1)`, denoted by $b5$, and with the precondition, `exp_at(bfm1,[n|posn],nil)`, denoted by $b2$, as a result of the implications 2 and ?? respectively from table 7–2. The remaining preconditions, for base case branch, are added to the list of unmatched preconditions

`exp_at(bfm1,[0|posn],bfm2)` denoted $b3$

`prim_rec(bfm2,n)` denoted $b4$

For the step case, there are several dependency links between the effects and various preconditions. These are shown in table 7–6

The remaining preconditions are added to the unmatched preconditions

`exp_at(sfm1,[0|posn],sfm2)` denoted $c3$

`prim_rec(sfm2,n)` denoted $c4$

`contain(sfm1,Ants)` denoted $d2$

---

[10] This implication is numbered 5a because the condition, `exp_at(New_state,Position,New_term)` is an expanded version of the condition `contain(New_state,New_term`, where `Position` denotes where in `New_state` the term `New_term` occurs.

Label/Effects	*Label/Preconditions*	*Imp*
*a*5 replace_all(var,[h1|t1], fm,sfm1)	*c*1 goal(sfm1)	3
	*c*2 exp_at(sfm1,[n|posn],[h1|t1])	5a
*a*6 include([h1,t1,ifm], hyplist1,hyplist2)	*d*5 hypothesis(ifm,hyplist2)	1
*a*4 replace_all(var,t1,fm,ifm)	*d*5 hypothesis(ifm,hyplist2)	4

Table 7–6: Dependency links between effects and preconditions at level 1

contain(sfm1,Cons)	denoted *d*3
contain(sfm1,Prec)	denoted *d*4
implication(ifm,Ants,Cons,Prec)	denoted *d*6

The back propagation process is applied to all the unmatched preconditions, from the base and step cases. This involves the effects of the operator, induct, *ie* *a*4, *a*5, *a*6 and *a*7. After the replacements of various arguments and re-expressing any meta-level arguments not dealt with by these effects, the resulting back propagated preconditions are:

exp_at(fm1,[0|posn],fm2)	denoted *b*3′ and *c*3′
prim_rec(fm2,n)	denoted *b*4′ and *c*4′
contain(fm1,Ants)	denoted *d*2′
contain(fm1,Cons)	denoted *d*3′
contain(fm1,Prec)	denoted *d*4′
implication(fm1,Ants,Cons,Prec)	denoted *d*6′

Figure 7–9 shows the representation of the completed plan at level 0 of the proof. The preconditions of this proof plan are the same as those for the IMPRESS proof plan presented in chapter 3, *viz*

goal(fm1)	denoted *a*1
hypothesis(var,hyplist1)	denoted *a*2

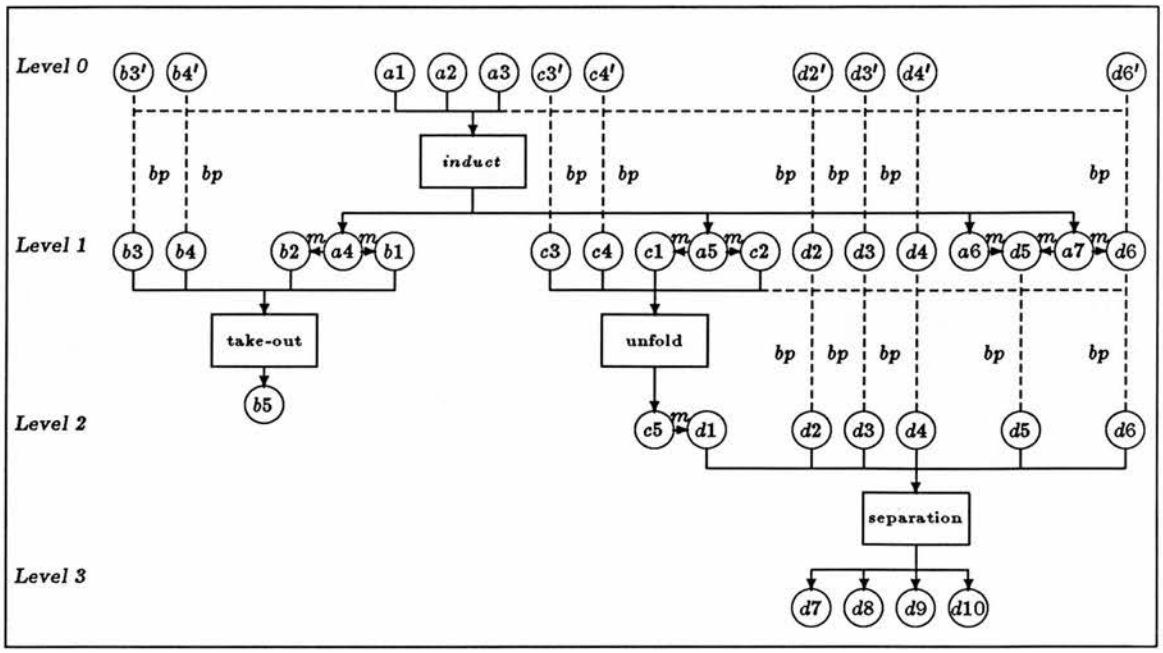


Figure 7–9: Representation of the completed IMPRESS plan

`contain(fm1,var)`	denoted $a3$
`exp_at(fm1,[0|posn],fm2)`	denoted $b3'$ and $c3'$
`prim_rec(fm2,n)`	denoted $b4'$ and $c4'$
`contain(fm1,Ants)`	denoted $d2'$
`contain(fm1,Cons)`	denoted $d3'$
`contain(fm1,Prec)`	denoted $d4'$
`implication(fm1,Ants,Cons,Prec)`	denoted $d6'$

This states that the IMPRESS proof plan is applicable to formulae which contain an implication between the two terms, `Ants` and `Prec`, and the term, `Conc`. These formulae must be primitively recursive in the argument position, `n`, of the term in position, `posn`, of the formula. This formula must also contain a variable, `var`, which, for the sake of the NuPRL environment, must already be an hypothesis.

It is possible to tighten up this specification by identifying that the variable, `var`, is in the primitively recursive position, `n` of the term in position, `posn`, of the formula. This is possible by replacing the precondition of the `induct` operator,

`contain(fm1,var)`, by the more expressive preconditions, `contain(fm1,exp1)` and `exp_at(exp1,[n|posn],var)`. The issue of making sure that the preconditions express the appropriate level of description of the formulae is discussed in the next section.

7.4 Benefits and Limitations of Extended-EBL

Extended-EBL has proved that it is able to learn proof plans. Two example proofs have been presented to the learning program and the resulting two proof plans learnt by Extended-EBL have been described above.

The first proof plan is generated from an existence proof for a function for inserting an element into a list. This is common for many proofs developed with the NuPRL proof environment. The second proof plan (IMPRESS) is applicable to formulae comprising simple implications between terms.

Extended-EBL relies on having a set of proof operators from which the proof operators required to prove the relevant formula can be identified and, of course, the proof itself. The identification process involves matching the appropriate preconditions and effects of the proof operator with the various states of the formula throughout the proof.

The generalisation of the proof plan involves back propagating the preconditions for each level of the proof. It has been shown how the role of the meta-language is vital not only for the back propagation process, but also for describing more general preconditions than those described in object-level terms.

However, the generality of the proof plan depends on the particular proof operators that were identified in the proof. If the proof comprises a laborious sequence of low-level proof steps, then the proof plan learned with Extended-EBL will comprise a corresponding sequence of low-level proof operators, together with preconditions describing the most general applicability of this sequence. The proof plan for the insert proof is of this type. It comprises a rather large

network of proof operators together with preconditions that show that it is applicable to formula comprising existential terms and other specific information about the formula.

On the other hand, the proof plan for the IMPRESS proof involves a much simpler network of proof operators such the preconditions for the applicability of this plan are less restrictive. Thus, although it is applicable only to formulae comprising simple implications, the proof plan learned is nonetheless applicable to all formulae of this type.

The meta-language plays a vital role within Extended-EBL, *eg* for enhancing the back propagation technique and for determining the specifications of missing operators. However, the cost of this is the initial development and refinement of the meta-language for the domain. This involves identifying not only the terms in the meta-language, but also the possible operators and their specifications, and the table of implications. The latter is essential for determining the contributions of operators within the plan, represented by dependency links between the preconditions and effects.

The two proof plans, described within this chapter, include similar operators requiring the same meta-level terms. Some of the implications for generating the dependency links are required for both proof plans. As more complex proof plans are learned more meta-level terms and implications will be required. Unfortunately, it is not possible to learn these with Extended-EBL.

Extended-EBL is very good for generalising control knowledge concerning the applicability and contribution of the proof operators within the plan. It is able to learn many types of structures of proof operators whether it is a simple sequence or tree. However, it is not able to generalise the network of proof operators further than is given by the structure of the proof. In order to do this Extended-EBL would have to generalise the proof first and then apply its current learning algorithm to extract the generalised proof plan.

Although, Extended-EBL is capable of learning the IMPRESS proof plan, it is not capable of learning the Boyer-Moore proof plan [Bundy 88b], which is

required for proving formulae comprising equivalence relations. The reason for this concerns its inability to deal with the `ripple-up` proof operator.

In all inductive proofs, the step case involves unfolding the step formula such that the inductive hypothesis may be used to terminate the proof. For formula involving simple implications the amount of unfolding required is minimal, often a simple application of the `unfold` operator, as is the case with the IMPRESS proof. However, for formulae comprising equivalence relations much more effort is required to unfold the step formula in order to make use of the inductive hypothesis. Often the part of the step formula necessary to match the inductive hypothesis is hidden very deeply within the formula. This is where the `ripple-up` operator fits in. The `ripple-up` proof operator involves many recursive calls on the `unfold` operator together with other operators for finding the next part of the formula to unfold.

However, the specifications for the `ripple-up` operator are difficult to represent. The indefinite repetition of the `unfold` operator is hard to capture. As a result, the back propagation process is not able to perform correctly because the state transitions between the proof state before and after the application of the `ripple-up` operator cannot be described well. Attempting to fit the `ripple-up` operator within the rest of the proof plan proves impossible, without describing the state transitions caused by the `ripple-up` operator.

7.5 Conclusions

This chapter consolidates the theoretical analysis provided in the first part of this thesis by describing the application of Extended-EBL to the task of learning proof plans. This domain demands the expressive control knowledge that is provided by the above learning program.

Chapter 5 showed how existing mechanisms within EBL could be integrated within a single program, Extended-EBL. The emphasis was on how the different types of control knowledge could be learned and combined with the structure of

the operators to form a generalised plan. However, it did not provide the details of the form and content of the control knowledge learned.

The analysis within this chapter, provides more flesh to the plans learned with Extended-EBL, by applying it to the task of learning proof plans. Proof plans require the more expressive control knowledge that can be learnt with Extended-EBL. The domain of theorem proving provides a rich source of knowledge which is essential for the EBL approach. Thus, this has certainly been a good domain in which to test the abilities of this powerful new learning technique.

Summary

In this chapter, the following has been presented and discussed.

- The interaction between object and meta-level knowledge provides a means of keeping the generalised plan in line with the detailed object-level description of the proof steps. This interaction has been found to be useful for learning the specifications of missing operators.

- Extended-EBL has been successfully applied to the task of learning proof plans for two proofs.

- The benefits and limitations of Extended-EBL are discussed in the context of the two proofs and proof plans presented in this chapter.

- Conclusions are presented about the applicability of Extended-EBL to the task of learning proof plans.

Chapter 8

Further Work

8.1 Introduction

The thesis, so far, has concentrated on my research into learning control knowledge within an EBL framework. I have presented a new technique, Extended-EBL, that incorporates the best features of current EBL approaches and I have applied this program to the task of learning proof plans. The result is an EBL technique with more expressive power for learning control knowledge from examples of solutions to problems.

In this chapter, some ideas and suggestions for further work are presented. These ideas are divided into two categories:

- some suggestions for further work that follow on *directly* from the research pursued within the thesis,

- other suggestions which could apply to other EBL approaches.

The two categories are denoted, *specific* and *general*, respectively. The specific suggestions refer to further extensions of the Extended-EBL program, taking into account more research and techniques from other areas in AI. The general suggestions are meant to reflect more ambitious goals, which would not only

benefit the Extended-EBL program, but also many other EBL programs. Because of the ambitious nature of these suggestions, they may not be as detailed as one might wish.

8.2 Specific Further Work

The following list contains suggestions for further work that continue the philosophy adopted within the thesis of extending the expressive power of the EBL approach for learning control knowledge.

- Learning partially-ordered plans
- Learning operator transformations

8.2.1 Learning partially-ordered plans

This thesis has shown how Extended-EBL is able to learn generalised plans for solving problems, in particular, proof plans for proving theorems.

Although, the control knowledge learnt aims to represent the most general conditions of applicability of the plan and its operators, the structure of the plan is fixed with respect to the training example, from which the generalised plan was learnt [1]. Thus, if the network of operators identified within the training example is structured as a linear sequence, a tree or a graph, then the structure of the plan is fixed with respect to that network of operators.

In most current planning systems, the generated plans permit *partially-ordered* operators [Sacerdoti 77,Tate 76,Vere 81,Currie 85,Wilkins 84]. This means

[1] It is possible for the plan to be patched, if it cannot be applied directly, by choosing alternative operators for particular steps of the plan. However, the general structure of the plan is fixed by learning.

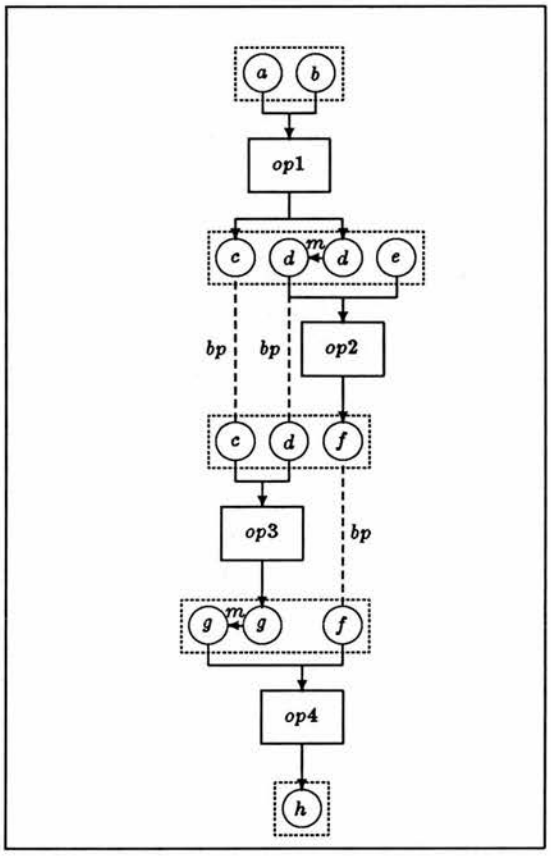


Figure 8–1: Solution to the problem involving the operators, *op*1-*op*4

that provided two operators do not interact with each other, by invalidating each other's specifications, then a *total-order* on these operators is unnecessary, and a *partial-order* is preferred. This allows much more flexibility in the execution of the plan, since the particular ordering of the application of relevant operators is constrained enough only for the plan to be successful.

Consider the solution to the problem as shown in figure 8–1. The solution comprises a sequence of four operators, $[op1, op2, op3, op4]$ together with their preconditions and postconditions [2]. Note that both *op*2 and *op*3 have pre-

[2]For the sake of simplicity, the specifications of the operators are trivial propositions, representing preconditions and postconditions.

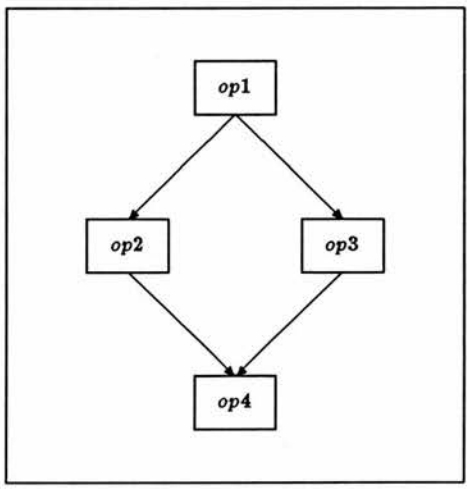


Figure 8–2: Partially-ordered structure for the operators, *op*1-*op*4

conditions which result from the postconditions of *op*1 and they also produce postconditions which are necessary as preconditions for *op*4.

The existing Extended-EBL program is certainly able to explain the contribution of each operator within the plan and identify that the condition *e* is a precondition for the entire plan, but, the structure of the learnt plan is restricted to a total-order on the operators, *ie* the sequence, $[op1, op2, op3, op4]$.

However, a more detailed look at the solution shows that it is not necessary for *op*2 to be applied before *op*3 for a solution to be generated. The sequence, $[op1, op3, op2, op4]$, would be equally successful. Thus, a partially-ordered plan, as shown if figure 8–2, would be more preferable and, indeed, more useful, and *should* be learnt.

Learning partially-ordered plans does not pose any great problems for the current version of the Extended-EBL program. The additional task for Extended-EBL involves determining which network of operators *must* be constrained to a total-order and which *may* be permitted a partial-order. Fortunately, the specifications of the operators are easily represented within Extended-EBL and so

manipulating these descriptions is possible [3]. The mechanism for determining whether the specification of two operators invalidate each other, *question-answering*, can be extracted from most current planning systems [Currie 85, Wilkins 84]. However, this process can only take place if all the specifications of operators involved in the plan are known. Thus, the specifications of missing operators must have been learned before the partial-ordering can be determined.

8.2.2 Learning operator transformations

This thesis has shown how the specifications of missing operators can be learned with the Extended-EBL program. Since, the effects of an operator represent the state transitions caused by the operator, then Extended-EBL tackles the task of learning about *operator transformations*. However, determining these specifications relies on having a table of implications, which records possible links between the effects of one operator and the preconditions of another. Unfortunately, this table of implications is not learned within Extended-EBL, but is already provided, such that in order to learn the specifications of missing operators relevant implications, which refer to these specifications, are required.

The PET program [Porter 84] addressed the same issue of learning about operator transformations or, what were termed, *relational models*. Relational models, in PET, map onto effects, in Extended-EBL. However, as discussed in chapters 2 and 6, the mechanism for learning these relational models was rather *ad hoc*. PET relied on two pattern matching processes at the object-level between parse trees representing descriptions of the problem states. The first matching process is required to propose a set of relational models (or effects) for each operator involved in the solution. The second is used to find which of these relational models were relevant to the solution by matching the terms of the relational models between these operators. This is analogous to determining

---

[3]Note that the extension to precondition analysis allowing postconditions to match non-adjacent preconditions is vital for partially-ordered plans.

the links between the effects of one operator and the preconditions of another. However, this second matching process is restricted to adjacent operators only. Thus, many relevant potential dependency links between relational models are not learned by PET. There is no such restriction with Extended-EBL, which is able to find dependency links between any of the operators involved in the plan.

PET went a little further in attempting to generalise the relational models by perturbing the arguments of the relations and checking whether they were still valid. The idea, of perturbing the training examples and checking whether the example is still a positive instance, is not new to learning, but has not be dealt with much in EBL.

The addition of this facility to the Extended-EBL program would be beneficial not only for providing more general specifications for known operators, but may also provide a better basis for learning the specifications of missing operators. There is no reason why this facility could not be incorporated into the existing framework.

To conclude, the Extended-EBL program provides a much better basis from which to tackle the task of learning about operator transformations than PET. The mechanism in Extended-EBL for generating the dependency links, the increased generality and expressiveness of the meta-level knowledge and the interaction of object and meta-level knowledge provide a good starting point for tackling this task.

8.3 General Suggestions for extending EBL

The following list contains suggestions for further work which are not only applicable to the work discussed in this thesis, but could also benefit most EBL approaches.

- Combining EBL and SBL techniques
- Evaluating concepts learnt during EBL

- Learning from failures

It is not intended that these suggestions provide detailed proposals for further work directly from the current thesis work. Instead, they reflect thoughts about some of the more ambitious goals for extending the EBL approach much further.

Together these suggestions for further work provide a good basis from which to extend the Extended-EBL program more towards the goal of a *learning apprentice* system [Mitchell 85]. Such a system would accumulate knowledge of its domain of interest by interaction with a teacher. It would reason about the concepts learnt in terms of the domain knowledge. It would assess the value of the new knowledge learnt, in the context of the current problem and other problems. It would learn from the failure to solve problems and explain the reason for failure. Thus, improving the overall capabilities of the learning apprentice system.

8.3.1 Combining EBL and SBL techniques

One of the initial aims of the EBL approach is to permit the refinement of existing concepts, when new information about the concept from other examples either enhances or contradicts the old concept. However, although this facility has been proposed in the past [DeJong 83,ORorke 83,Niblett 87], it has not been dealt with much by EBL researchers, except recently [Lebowitz 86,Pazzani 87].

It has been shown within this thesis, how the EBL approach proves successful in domains where there is a strong theory that models its behaviour. The SBL approach does not require a strong domain theory and learns by incrementally inducing and refining its concepts from several examples, over time. Combining the explanatory facilities of an EBL approach with the pattern matching, inductive approach of the SBL, could provide a better basis from which to develop a learning apprentice system that evolves and steadily improves its performance.

For EBL techniques, good representation of domain knowledge is essential. So far, the meta-language proposed, in this thesis, has been a rather *flat* language,

with no notion of a hierarchy of meta-level terms. Most other machine learning techniques, excluding EBL, certainly make use of a hierarchical generalisation language [Winston 75,Mitchell 82a], for generalising features common to several instances of the concept to be learned. In the same way, meta-level terms could be generalised with reference to a meta-level generalisation hierarchy.

The addition of an hierarchical meta-language to the Extended-EBL could provide a better basis for a combined EBL/SBL system than any of the other proposals which rely on object-level descriptions of the control knowledge. It would permit plans to be described with more general control knowledge and at many levels of abstraction. This would allow more flexibility in plan execution [4]. Thus, Extended-EBL would be able to refine and generalise further the existing learnt plan when new information, from other similar problems, is provided.

8.3.2 Evaluating concepts learnt during EBL

Research in evaluating concepts learnt during EBL has already been discussed in chapter 2. Both the PRODIGY [Minton 87] and MetaLEX [Keller 87] programs incorporated an analysis of the *utility* of the learned concept, within the learning process. The utility analysis provides a means of improving problem solving performance by removing inefficient operators from the problem solving system. It also provides a means of stopping the learning process by setting performance objectives, based on the utility analysis.

Incorporating some utility analysis within Extended-EBL would require a major effort. In particular, it would require the development of a meta-language dedicated to representing notions of efficiency and performance of the learnt plans. Mechanisms would need to be provided within Extended-EBL for testing the learnt plans on suitable sets of examples and for modifying these sets when more complex examples can be solved. The meta-language would reflect the

---

[4]Knowledge about possible state transitions at a more abstract level could also provide a more principled starting point for learning about operator transformations.

criteria for measuring performance of the learnt plans. Such an endeavour would prove invaluable for both problem solving and learning systems, and would lead closer to the notion of learning apprentice systems.

8.3.3 Learning from failures

The most challenging problems are always left until the end. Learning from failures in solving problems, is definitely a very challenging area for machine learning.

Certainly for a learning apprentice system, the ability to learn from mistakes and blind alleys in problem solving would be of great benefit. Indeed, most inductive learning techniques make use of negative examples to constrain over-generality of the concept to be learnt. But these negative examples or *near misses* are carefully selected to have only one negative feature, such that the discrimination with the target concept is straightforward.

However, the EBL approach would require the failure to be explained in terms of the domain knowledge. Determining exactly what caused the failure to proceed further in a solution is a non-trivial task. The particular cause of the failure may have arisen several steps back in the solution, although, the actual failure to proceed further may not occur until later on in the solution.

Even though Extended-EBL has a good mechanism for managing the dependencies between operators, finding the culprit for a particular failure is going to be very difficult. There have been some recent attempts at learning from failures [Hammond 86,Hammond 87,Hall 86], but the research is very much at an early stage. This promises to be a very hot topic for EBL research for the next few years.

Chapter 9

Conclusions

9.1 Overall Contributions of the Thesis

In this thesis, the following contributions are made to the task of learning control knowledge and extending the EBL approach:

- Extended-EBL can learn more expressive control knowledge than any of the individual EBL techniques, from which it is derived,
- Extended-EBL has been successfully applied to the task of learning proof plans. This domain requires the more expressive control knowledge that can be learned with the Extended-EBL program.
- Based on the discussions in chapter 8, Extended-EBL provides a firmer foundation from which to further extend the EBL approach.

Together these provide a significant contribution to machine learning research, and, in particular, to the area of explanation-based learning. The success of the Extended-EBL for learning control knowledge confirms the EBL approach as an important machine learning technique.

The thesis discusses research in the mainstream of machine learning. It addresses issues that are of current interest and important for the application of machine learning to the next stage of knowledge-based system development.

9.2 Extended-EBL

Extended-EBL shows how existing EBL techniques can be incorporated within a single program, such that much more expressive control knowledge can be learnt from single examples of successful solutions to problems.

The success of Extended-EBL relies on the following factors:

- the *classification* of different types of control knowledge;
- the *role* of the meta-level knowledge;
- the description of effects as *state transitions*;
- the ability to learn the specifications of *missing* operators;
- *extending* previous EBL techniques.

9.2.1 Classification of types of control knowledge

The survey in chapter 2 motivated the development of the extended EBL approach, by identifying the different types of control knowledge that were learnt by previous EBL techniques. These are listed below

- Contribution of each operator within the plan
- Applicability of each part of the plan
- Explicit structure of the plan

The classification of the different types of control knowledge [1] divided the learning task into much more manageable sub-tasks. This allowed a particular technique to be associated with the achievement of a particular sub-task.

9.2.2 Role of the meta-level knowledge

Representing the specifications of the operators at the meta-level means that the above types of control knowledge can be expressed in the most general terms possible. Chapter 5, 6 and 7 show how these specifications are manipulated to provide the relevant control knowledge. Thus, the search for the most appropriate plan or operator is improved, and, if a learnt plan cannot be fully applied, then the plan can be repaired, without necessarily invalidating the rest of the plan.

However, the cost of this is the initial development and refinement of the meta-language for the domain. This involves identifying not only the terms in the meta-language, but also the possible operators and their specifications, and the table of possible implications. Although, the development of a meta-language is costly in time and effort, the experiences with PRESS/LP and more recently with CLAM/OYSTER [Bundy 88d,Bundy 88e] have shown that it is worthwhile. The latter two programs, both developed within Bundy's Mathematical Reasoning Group, have proved to be useful testbeds for experimenting with meta-theories for proof plans.

Experience with these programs also suggests that there is no need for a totally new or special meta-theory for each domain – many of the meta-level terms are common across plans for different domains. For instance, the meta-level term in LP for identifying the *occurrence* of the unknown in the equation to be solved is similar to the meta-level term, described in chapters 3, 6 and 7, for showing that a formula *contains* a certain variable or expression. However,

---

[1] Although, the classification is by no means comprehensive, it is unique as far as I know.

there is a need for a mechanism for updating and adding new meta-level terms to the theory, *eg* inverse resolution [Muggleton 88].

9.2.3 Effects as state transitions

In chapter 3, a distinction was made between the meta-language for describing the preconditions and effects of the operators. The preconditions were represented by meta-level terms describing *properties* of the proof state, and the effects by terms describing *state transitions* [2]. Although, this distinction is not essential for the domain itself, it provides valuable information about the operator transformations which were found to be essential for extending the ability to learn the specifications of missing operators.

As a result, the original Extended-EBL program, from chapter 5 was revised to make use of this extra knowledge. The main revisions were the following

- the matching process was enhanced from a trivial unification process to one involving inference between the effects of one operator and the preconditions of another

- the effects of an operator provide knowledge about the state transitions produced by that operator, which is used to enhance the back propagation process

- the interaction between object and meta-level knowledge provides a means of dealing with indirect changes to partial states of the proof, which are not reflected in the effects of an operator.

The effects provide a lot of valuable information about the state transitions caused by operators, but do not capture all the repercussions of these trans-

---

[2] In PRESS and LP only properties of the problem state were represented by meta-level terms.

formations. Thus, the effects cannot be relied upon to provide *complete* frame axioms.

9.2.4 Learning specifications

Revisions to Extended-EBL in chapter 6 show how the task of learning the specifications of missing operators can be improved.

The original precondition analysis technique was able to learn the specifications of missing operators by manipulating meta-level terms alone. Unfortunately, because of the fear of over-generalising these specifications, the approach was far too conservative. As a result, the specifications were often incomplete.

Suggestions made in chapter 4, extended the technique a little by representing the conditions that are deleted by the missing operator, as preconditions of that operator. However, in chapter 5, the problem of determining the specifications of two missing operators shows how inadequate these extensions are. Although, Extended-EBL, as described in chapter 5, is able to learn control knowledge which spans many operators in the plan, it is unable to associate the correct specifications to solutions involving more than one missing operator.

Chapter 6 shows how the interaction between the object and meta-level knowledge, together with the description of the effects of operators as state transitions, provide the means of learning the *specifications* of missing operators, no matter how many are missing, within reason [3]. The ability to learn the specifications of operators, in solutions with more than one missing operator, goes much further than any existing EBL technique. However, the ability to learn *complete* specifications of missing operators is beyond the reach of the current approach.

---

[3]If all the operators are unknown to the learning system, then no learning can take place, since there is no domain knowledge to explain the solution and guide the generalisation process.

9.2.5 The extended EBL approach

The research methodology adopted within this thesis involves building upon previous work, by rationally reconstructing the most important features of past efforts, and extending them further [4].

Extended-EBL is able to learn all three of the different types of knowledge discussed in chapter 2.

The applicability of each part of the plan can be learnt due to the back propagation process, as described in chapters 5 and 6. This process is more expressive than the constraint back propagation technique [Mitchell 83b].

The contribution of each operator within the plan and the explicit structure of the plan can be learnt due to the use of meta-level knowledge and a dependency graph. The resulting control knowledge is better than can be achieved with either the precondition analysis technique [Silver 84] or MACROPS [Fikes 72].

The result of the extended EBL approach is that more expressive control knowledge can be learnt than by applying the individual EBL techniques. This is primarily because it is the *rationale* behind each technique, rather than the techniques themselves which are incorporated within Extended-EBL. Thus, it can be said that the contribution of Extended-EBL is greater than the sum of its parts.

---

[4]This research methodology is standard in most scientific disciplines [Kuhn 70], but, has often been ignored within AI. For some odd reason, there is an ill-conceived notion that "if a topic has already been tackled by someone, then another person's attempt to improve upon that work, is often not considered to be original". This thesis attempts to contradict this ill-conceived reasoning and presents a thesis based upon good scientific methodology.

9.3 Learning Proof Plans

Chapters 6 has shown how Extended-EBL can be successfully applied to the task of learning proof plans.

The task of learning proof plans provides a domain which requires the more expressive control knowledge that can be learnt with the Extended-EBL program. Although, the representation and use of proof plans is still very much in its infancy, the area of theorem proving has been the subject of intensive research, for many years, by mathematicians and logicians. As a result, there is a great deal of useful domain knowledge, which is essential for an EBL approach to be successful.

The Extended-EBL program can be applied to domains which make use of an explicit representation of the state transitions of its operators, or, to those where the domain knowledge is in terms of properties of the problem states alone. Chapters 5, 6 and 7 have shown how Extended-EBL can deal with both.

9.4 Foundation for further EBL development

Chapter 8 has made various suggestions for further work. All of the suggestions involve extending the power of the EBL approach even further. Some of the suggestions are generally applicable to other EBL approaches, but others follow *directly* from this thesis work. These are

- Learning partially-ordered plans
- Learning operator transformations
- Hierarchical meta-language

Performing these tasks would require even more expressive control knowledge. However, Extended-EBL provides the foundation from which to tackle these tasks.

Bibliography

[Benjamin 87] D.P. Benjamin. Learning strategies by reasoning about rules. In J. McDermott, editor, *Proceedings of the tenth IJCAI*, pages 256–259, International Joint Conference on Artificial Intelligence, 1987.

[Boyer 79] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.

[Bundy 81] A. Bundy and B. Welham. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212, 1981. Also available as DAI Research Paper 121.

[Bundy 84] A. Bundy, editor. *Catalogue of Artificial Intelligence Tools*, Springer-Verlag, 1984. Second Edition.

[Bundy 86] A. Bundy and L.A. Wallen. *Proving properties of logic programs: summary of progress*. Research paper 312, Dept. of Artificial Intelligence, Edinburgh, September 1986.

[Bundy 88a] A. Bundy. The use of explicit plans to guide inductive proofs. In *9th Conference on Automated Deduction*, pages 111–120, Springer-Verlag, 1988. Longer version available as DAI Research Paper No. 349.

[Bundy 88b] A. Bundy. *The Use of Explicit Plans to Guide Inductive Proofs*. Research Paper 349, Dept. of Artificial Intelligence,

Edinburgh, 1988. Short version published in the proceedings of CADE-9.

[Bundy 88c] A. Bundy and L.S. Sterling. Meta-level inference: two applications. *Journal of Automated Reasoning*, 4(1):15–27, 1988. Also available from Edinburgh, Dept. of AI, as Research Paper No. 273.

[Bundy 88d] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. *Experiments with Proof Plans for Induction.* Research Paper 413, Dept. of Artificial Intelligence, Edinburgh, 1988. Submitted to JAR.

[Bundy 88e] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. *A rational reconstruction and extension of recursion analysis.* Research Paper 419, Dept. of Artificial Intelligence, Edinburgh, 1988. To appear in the proceedings of IJCAI-89.

[Constable 86] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Currie 85] K. Currie and A. Tate. O-Plan: Control in the open planning architecture. In *Proceedings of BCS Expert Systems 85*, Cambridge University Press, 1985.

[Darlington 81] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(3):1–46, August 1981.

[Davis 80] R. Davis. Meta-rules: reasoning about control. *Artificial Intelligence*, 15:179–222, 1980.

[DeJong 83] G. DeJong. Acquiring schemata through understanding and generalizing plans. In A. Bundy, editor, *Proceedings of the*

Eighth IJCAI, pages 462–464, International Joint Conference on Artificial Intelligence, Karlsruhe, Germany, 1983.

[DeJong 86] G. DeJong and R. Mooney. Explanation-based learning: an alternate view. *Machine Learning*, 1(2):145–176, 1986.

[Desimone 87] R.V. Desimone. Learning control knowledge within an explanation-based learning framework. In I. Bratko and N. Lavrač, editors, *Progress in Machine Learning – Proceedings of 2nd European Working Session on Learning, EWSL-87, Bled, Yugoslavia*, Sigma Press, May 1987. Also available as DAI Research Paper 321.

[Desimone 89] R.V. Desimone. Explanation-Based Learning of Proof Plans. In Y. Kodratoff and A. Hutchinson, editors, *Machine and Human Learning*, Kogan Page, 1989. Also available as DAI Research Paper 304. Previous version in proceedings of EWSL-86.

[Dijkstra 76] E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Fikes 71] R.E. Fikes, , and N.J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Fikes 72] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Follett 84] R. Follett. Combining program synthesis with program analysis. In A Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 91–107, MacMillan, 1984.

[Good 77] I.J. Good. Dynamic probability, computer chess and the measurement of knowledge. In *Machine Intelligence 8*, pages 139–150, Halstead and Wiley, 1977.

[Hall 86] R. Hall. Learning by failing to explain. In *Proceedings of AAAI-86*, pages 568–573, American Association for Artificial Intelligence, 1986.

[Hammond 86] K.J. Hammond. Learning to anticipate and avoid planning problems through the explanation of failures. In *Proceedings of AAAI-86*, pages 556–560, American Association for Artificial Intelligence, 1986.

[Hammond 87] K.J. Hammond. Learning and reusing explanations. In P. Langley, editor, *Proceedings of the 4th International Machine Learning Workshop*, pages 141–147, Morgan Kaufmann, 1987.

[KedarCabelli 87] S. Kedar-Cabelli and L.T. McCarty. Explanation-based generalization as resolution theorem proving. In P. Langley, editor, *Proceedings of the 4th International Machine Learning Workshop*, pages 383–389, Morgan Kaufmann, University of California, Irvine, CA, June 1987. Also available as Research Report No. ML-TR-10, Lab. for Computer Science Research, Hill Center for Mathematical Sciences, Rutgers University, New Brunswick, New Jersey.

[Keller 80] R.M. Keller. Learning by re-expressing concepts for efficient recognition. In *Proceedings of AAAI-80*, pages 182–186, American Association for Artificial Intelligence, 1980.

[Keller 87] R.M. Keller. Concept learning in context. In P. Langley, editor, *Proceedings of the 4th Machine Learning Workshop*, pages 91–103, Morgan Kaufmann, 1987.

[Kling 71] R.E. Kling. A paradigm for reasoning by analogy. *Artificial Intelligence*, 2, 1971.

[Kowalski 79a] R. Kowalski. Algorithm = Logic +Control. *Communications of ACM*, 22:424–436, 1979.

[Kowalski 79b] R. Kowalski. *Logic for Problem Solving. Artificial Intelligence Series*, North Holland, 1979.

[Kuhn 70] T.S. Kuhn. *The Structure of Scientific Revolutions.* Volume 2 of *Foundations of the Unity of Science*, University of Chicago Press, 1970. Second Edition, Enlarged.

[Lebowitz 86] M. Lebowitz. Not the path to perdition: The utility of Similarity-Based Learning. In *Proceedings of AAAI-86*, pages 533–537, American Association for Artificial Intelligence, 1986.

[MartinLof 79] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hannover, August 1979. Published by North Holland, Amsterdam. 1982.

[McCarthy 69] J. McCarthy and P Hayes. Some philsophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, Edinburgh University Press, 1969.

[Michalski 77] R.S. Michalski and P.G. Negri. An experiment on inductive learning in chess end-games. In *Machine Intelligence 8*, pages 175–192, Halstead and Wiley, 1977.

[Michalski 82] R.S. Michalski, J.H. Davis, V.S. Bisht, and J.B. Sinclair. Plant/ds: an expert consulting system for the diagnosis of

soybean diseases. In *Proceedings of ECAI-82*, pages 139–140, European Conference on Artificial Intelligence, 1982.

[Michalski 83] R.S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161, 1983.

[Minsky 75] M. Minsky. A framework for representing knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, McGraw-Hill, 1975.

[Minton 84] S. Minton. Constraint-based generalization: learning game-playing plans from single examples. In R. Brachman, editor, *Procs. of AAAI-84*, pages 251–254, American Association for Artificial Intelligence, 1984.

[Minton 87] S. Minton, J.G. Carbonell, O. Etzioni, C.A. Knoblock, and D.R. Kuokka. Acquiring effective search control rules: explanation-based learning in the PRODIGY system. In P. Langley, editor, *Proceedings of the 4th Machine Learning Workshop*, pages 122–133, Morgan Kaufmann, 1987.

[Mitchell 81] T.M. Mitchell, P. E. Utgoff, B. Nudel, and R. Banerji. Learning problem-solving heuristics through practice. In *Proceedings of IJCAI-81*, pages 127–134, International Joint Conference on Artificial Intelligence, 1981.

[Mitchell 82a] T.M. Mitchell. Generalisation as search. *Artificial Intelligence*, 18:203–227, 1982.

[Mitchell 82b] T.M. Mitchell. *Toward Combining Empirical and Analytical Methods For Inferring Heuristics.* Technical Report LCSR-TR-27, Laboratory for Computer Science Research, Rutgers University, 1982.

[Mitchell 83a] T.M. Mitchell. Learning and problem solving. In A. Bundy, editor, *Proceedings of the Eighth IJCAI*, pages 1139–1151, International Joint Conference on Artificial Intelligence, Karlsruhe, Germany, 1983.

[Mitchell 83b] T.M. Mitchell, P. E. Utgoff, and R. Banerji. Learning by experimentation: acquiring and modifying problem-solving heuristics. In *Machine Learning*, pages 163–190, Tioga Press, 1983.

[Mitchell 85] T.M. Mitchell, S. Mahadevan, and L.I. Sternberg. LEAP: A learning apprentice for VLSI design. In *Proceedings of the ninth IJCAI*, pages 573–580, International Joint Conference on Artificial Intelligence, 1985.

[Mitchell 86] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation-based generalization: a unifying view. *Machine Learning*, 1(1):47–80, 1986. Also available as Tech. Report ML-TR-2, SUNJ Rutgers, 1985.

[Mooney 85] R. Mooney and G. DeJong. Learning schemata for natural language processing. In Mackworth A., editor, *Proceedings of the ninth IJCAI*, pages 681–687, IJCAI, 1985.

[Mooney 86] R. Mooney and S. Bennett. A domain independent explanation-based generalizer. In *Proceedings of AAAI-86*, pages 551–555, Morgan Kaufmann, 1986.

[Mostow 81] D.J. Mostow. *Mechanical transformation of task heuristics into operational procedures.* PhD thesis, Carnegie-Mellon University, 1981.

[Muggleton 88] S. Muggleton. A strategy for constructing new predicates in first-order logic. In *Proceedings of the Third European Working Session on Learning*, Pitman Publishing, 1988.

[Newell 63] A. Newell and H.A. Simon. The Logic Theory machine. In E.A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, McGraw-Hill, 1963.

[Newell 72] A. Newell and H.A. Simon. *Human problem solving.* Prentice-Hall, 1972.

[Newell 73] A. Newell. Production systems; a model of control structures. In W.G. Chase, editor, *Visual Information Processing*, Academic Press, 1973.

[Niblett 87] T. Niblett. Learning new concepts. In *Proceedings of Colloquium on Machine Learning*, Institution of Electrical Engineers, 1987.

[Nilsson 80] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Pub. Co., Palo Alto, California, 1980.

[ORorke 83] P. O'Rorke. Reasons for belief in understanding: applications of non-monotonic dependencies in story processing. In *Proceedings of AAAI-83*, pages 306–309, American Association for Artificial Intelligence, 1983.

[Pazzani 87] M.J. Pazzani. Inducing causal and social theories: A prerequisite for Explanation-Based Learning. In P. Langley, editor, *Proceedings of the Fourth Machine Learning Workshop*, pages 230–241, Morgan Kaufmann, 1987.

[Porter 84] B. Porter and D. Kibler. Learning operator transformations. In R. Brachman, editor, *Proceedings of AAAI-84*, pages 278–282, American Association for Artificial Intelligence, 1984.

[Prieditis 87] A.E. Prieditis and J. Mostow. PROLEARN: Towards a Prolog interpreter that learns. In K. Forbus and H. Shrobe, editors, *Proceedings of AAAI-87*, pages 494–498, Morgan Kauf-

mann, Seattle, Washington, July 1987. Also available as Research Report No. ML-TR-13, Lab. for Computer Science Research, Hill Center for Mathematical Sciences, Rutgers University, New Brunswick, New Jersey.

[Puget 87] J-F. Puget. Goal regression with opponent. In I. Bratko and Lavrăc, editors, *Progress in Machine Learning: Proceedings of the 2nd European Working Session on Learning*, pages 121–137, Sigma Press, 1987.

[Quillian 68] M.R. Quillian. Semantic memory. In M. Minsky, editor, *Semantic Information Processing*, pages 227–259, MIT Press, 1968.

[Raphael 68] B. Raphael. A computer program for semantic information retrieval. In M. Minsky, editor, *Semantic Information Processing*, MIT Press, 1968.

[Sacerdoti 77] E.D. Sacerdoti. *A Structure for Plans and Behaviour. Artificial Intelligence Series*, North Holland, 1977. Also as SRI AI Technical note number 109, August 1975.

[Schubert 76] L.K. Schubert. Extending the expressive power of semantic networks. *Artificial Intelligence*, 7:89–124, 1976.

[Segre 85] A.M. Segre. Explanation-Based manipulator learning. In *Proceedings of the Third Machine Learning Workshop*, Skytop, 1985.

[Shavlik 85] J.W. Shavlik. Learning classical physics. In *Proceedings of the Third Machine Learning Workshop*, Skytop, 1985.

[Silver 83] B. Silver. Learning equation solving methods from examples. In A. Bundy, editor, *Proceedings of the Eighth IJCAI*, pages 429–431, International Joint Conference on Artificial

Intelligence, 1983. Also available from Edinburgh as Research Paper 184.

[Silver 84] B. Silver. Precondition analysis: learning control information. In *Machine Learning 2*, Tioga Publishing Company, 1984.

[Silver 85] B. Silver. *Meta-level inference: Representing and Learning Control Information in Artificial Intelligence.* North Holland, 1985. Revised version of the author's PhD thesis, DAI 1984.

[Stefik 81] M. Stefik. Planning and meta-planning (MOLGEN: part 2). *Artificial Intelligence*, 16:141–170, 1981.

[Sterling 82] L. Sterling, A. Bundy, L. Byrd, R. O'Keefe, and B. Silver. Solving symbolic equations with PRESS. In J. Calmet, editor, *Computer Algebra, Lecture Notes in Computer Science No. 144.*, pages 109–116, Springer Verlag, 1982. Also available from Edinburgh as Research Paper 171.

[Tate 76] A. Tate. *Project Planning Using a Hierarchic Non-Linear Planner.* Research Report 25, Dept. of Artificial Intelligence, Edinburgh, 1976.

[Utgoff 84] P.E. Utgoff. *Shift of Bias for Inductive Concept Learning.* PhD thesis, Rutgers University, October 1984.

[van Harmelen 87] F. van Harmelen. *A Classification of Meta-level Architectures.* DAI Research Paper 297, Dept. of Artificial Intelligence, Edinburgh, 1987. To appear in: 'Logic-Based Knowledge Representation', Reichgelt, H., Jackson, P. and van Harmelen, F. (eds), published by MIT Press 1988.

[Vere 81] S.A. Vere. Planning in Time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):246–267, 1981.

[Waldinger 77] R. Waldinger. *Achieving Several Goals Simultaneously*, chapter 6, pages 94–138. Volume 8 of *Machine Intelligence*, Halstead and Wiley, New York, 1977.

[Wallen 83a] L.A. Wallen. *Towards the Provision of a Natural Mechanism for Expressing Domain-Specific Global Strategies in General Purpose Theorem-Provers.* Research Paper 202, Dept. of Artificial Intelligence, Edinburgh, September 1983.

[Wallen 83b] L.A. Wallen. *Using Proof Plans to Control Deduction.* Research Paper 185, Dept. of Artificial Intelligence, Edinburgh, February 1983.

[Wallen 86] L.A. Wallen. Generating connection calculi from tableau- and sequent-based proof systems. In A.G. Cohn and J.R. Thomas, editors, *Artificial Intelligence and its Applications*, pages 35–50, John Wiley & Sons Ltd., 1986. Also in Proc. of AISB-85, ed. P. Ross.

[Wallen 87a] L.A. Wallen. *Matrix proof methods for first-order modal logics.* Research paper , Dept. of Artificial Intelligence, Edinburgh, 1987. (To appear).

[Wallen 87b] L.A. Wallen and G.V. Wilson. A computationally efficient proof system for S5 modal logic. In *Proceedings of AISB-87*, Wiley & Sons, 1987. (To appear.).

[Wilkins 84] D.E. Wilkins. Domain independent planning: Representation and plan generation. *Artificial Intelligence*, 22, 1984.

[Winston 75] P. Winston. Learning structural descriptions from examples. In P.H. Winston, editor, *The psychology of computer vision*, McGraw Hill, 1975.

Appendix A

The Full Insert Proof

A.1 Introduction

The aim of this appendix is to present the full structure of the proof of the insert function that was partially examined in chapter 3. For clarity, the *type* information is removed from the proof as well as the relevant well-formedness proofs that relate to these typed goals.

The main part of the proof has already been described in chapter 3, but is included again here for clarity.

A.2 The Main Proof

The application of the first proof operator, `parm_tac`, removes all the outermost universal quantifiers from the goal, represented by the expression

$$\gg \forall a.\forall x.\exists z.I(a,x,z)$$

The previously quantified parameters are now represented as the hypotheses, [1,2], of the new sub-goal to be proved, $\exists z.I(a,x,z)$. Refer to proof on page 221.

The next operator, `induct`, splits the proof tree into two sub-goals, one which deals with the base case, represented by the expression, $\exists z.I(a,nil,z)$, and the step case, represented by the expression, $\exists z.I(a,u\cdot x1,z)$ [1]. These two sub-goals are produced by replacing the induction variable, represented by the list, x, with the base and step values, nil and $u\cdot x1$ respectively. In addition, for the step case, some hypotheses are added to the hypothesis list. The two hypotheses, u and $x1$ are essential for proving the well-formedness of the expression, $I(a,u\cdot x1,z)$. However, the other hypothesis, $\exists z.I(a,x1,z)$ represents the induction hypothesis, which is required later on in the proof.

For the base case, the application of the next operator, `instantiate_existential_goal`, involves instantiating the existential variable, z, with the appropriate value, $a\cdot nil$ [2]. This leaves the expression $I(a,nil,a\cdot nil)$ still to be proved. The rest of this branch of the proof tree involves further decomposition of the expression until a termination node is reached. This occurs when the goal

---

[1] The $\cdot$ connective, denotes a *cons* binary operator, such that the list $u\cdot x1$ has u at the head and $x1$ at the tail of the list.

[2] This value is appropriate because the result of inserting a into the empty list, nil, is the list, $a\cdot nil$. Since NuPRL is an interactive proof development system and not an automatic theorem prover, this value has to be provided by the user.

to be proved matches one of the hypotheses of the hypothesis set or matches some previously proved lemma.

For the step case, the next two operators, `instantiate_existential_hypothesis` and `instantiate_existential_goal`, perform similiar tasks. The existential variable, z, in both the induction hypothesis, $\exists z.I(a, x1, z)$ and the step goal, $\exists z.I(a, u \cdot x1, z)$ is instantiated by $z1$ and $u \cdot z1$ respectively. This keeps the description of both the new induction hypothesis, $I(a, x1, z1)$ and the new step goal, $I(a, u \cdot x1, u \cdot z1)$ at the same nested level. The rest of the step case proof proceeds until the tree is terminated. Note that this involves the use of the revised induction hypothesis.

$\gg \forall a.\forall x.\exists z.I(a,x,z)$

by parm_tac

1. a
2. x

$\gg \exists z.\ I(a,x,z)$

by induct x new $p,u,x1$

$1-2$

$\gg \exists z.\ I(a,nil,z)$

by instantiate_existential_
goal $a \cdot nil$

$1-2$

$\gg I(a,nil,a \cdot nil)$

•
•
•

Cont'd on page 224

$1-2$

3. u
4. $x1$
5. $\exists z.\ I(a,x1,z)$

$\gg \exists z.\ I(a,u \cdot x1,z)$

by instantiate_existential_
hypothesis p new $z1, ih$

$1-5$

6. $z1$
7. $I(a,x1,z1)$

$\gg \exists z.\ I(a,u \cdot x1,z)$

by instantiate_existential_
goal $u \cdot z1$

$1-7$

$\gg I(a,u \cdot x1,u \cdot z1)$

•
•
•

Cont'd on page 227

A.2.1 Base Case Proof Cont'd

The term, $I(a, x, z)$ represents an expression for inserting an element, a, into the list, x, resulting in the new list, z. It can be decomposed further into the following expression

$$I(a, x, z) \equiv (\forall e.mem(e, z) \Leftrightarrow e = a \mid mem(e, x))$$

which states that for all elements e, if e is a member of the list z, then either it is equal to the element a or else it is a member of the list x. This expands further the specification of the insert predicate, I. Refer to page 224.

The application of the next operator, `parm_tac`, strips away the universal quantifier, $\forall e$, and adds it to the hypothesis list for the sub-goal

$$\gg mem(e, a \cdot nil) \Leftrightarrow e = a \mid mem(e, nil)$$

The operator, `remove_equivalence`, splits the equivalence connective into a conjunction of two implications, thus producing two sub-goals to prove

$$\gg mem(e, a \cdot nil) \rightarrow e = a \mid mem(e, nil)$$
$$\gg e = a \mid mem(e, nil) \rightarrow mem(e, a \cdot nil)$$

The proof of both sub-goals involves the application of the same operators but in a different order.

The proof of the left sub-goal involves the application of the operator, `unfold`, followed by the operator, `remove_implication` and, finally, the operator, `hypothesis`.

The `unfold` operator, replaces the *lhs* of the implication in the sub-goal with the unfolded form, $e = a \mid mem(e, nil)$. The `remove_implication` operator, assumes the *lhs* of the implication by placing it in its list of hypotheses and attempts to prove the *rhs*. The application of the `hypothesis` operator, notes that there is an hypothesis which matches the current sub-goal, thus terminating the proof.

The proof of the right sub-goal is very similar to that of the left goal, except that the `remove_implication` operator is applied first, then the `unfold` operator, followed by the `hypothesis` operator.

Unfortunately, NuPRL, in its current form, has no way of recognising that the sub-goal containing the equivalence connective actually represents the *unfold* axiom for $mem(e, a \cdot nil)$.

$1-2$
$\gg\ I(a, nil, a \cdot nil)$

by parm_tac

$1-2$
3. e
$\gg\ mem(e, a \cdot nil) \Leftrightarrow e = a \mid mem(e, nil)$

by remove_equivalence

$1-3$
$\gg\ mem(e, a \cdot nil) \rightarrow$
$\quad e = a \mid mem(e, nil)$

by unfold

$1-3$
$\gg\ e = a \mid mem(e, nil) \rightarrow$
$\quad e = a \mid mem(e, nil)$
by remove_implication

$1-3$
4. $e = a \mid mem(e, nil)$
$\gg\ e = a \mid mem(e, nil)$

by hypothesis

□

$1-3$
$\gg\ e = a \mid mem(e, nil) \rightarrow$
$\quad mem(e, a \cdot nil)$

by remove_implication

$1-3$
4. $e = a \mid mem(e, nil)$
$\gg\ mem(e, a \cdot nil)$

by unfold

$1-4$
$\gg\ e = a \mid mem(e, nil)$

by hypothesis

□

A.2.2 Step Case Proof Cont'd

The next two operators, `parm_tac` and `remove_universal_hypothesis`, strip away the universal quantifier, $\forall e$, from both the goal, $I(a, u \cdot x1, u \cdot z1)$ and the induction hypothesis, ih. This results in the following sub-goal

$$mem(e, u \cdot z1) \Leftrightarrow e = a \mid mem(e, u \cdot x1)$$

and a revised form of the inductive hypothesis that is at the same nested level. Refer to page 227.

The next two operators, `remove_equivalence_hypothesis` and `remove_equivalence`, again perform the same operations on both the goal and the hypothesis. The resulting two sub-goals

$$\gg mem(e, u \cdot z1) \rightarrow e = a \mid mem(e, u \cdot x1)$$
$$\gg e = a \mid mem(e, u \cdot x1) \rightarrow mem(e, u \cdot z1)$$

represent the conjunction of the two implications involved in the equivalence expression. In order to keep the inductive hypothesis in line with the step goal, two new hypotheses, 11 and 12, are added to the hypothesis list for each of these sub-goals .

The proof of the right sub-goal is not described further in this appendix, because it involves the same operators as in the proof of the left sub-goal, but in a different order [3].

The proof of the left sub-goal involves the application of the `unfold` operator followed by the `remove_implication` operator. However, the `hypothesis` operator cannot be applied, since there is no hypothesis which matches the goal to be proved.

---

[3]Note that so far the applications of the operators are very similar to those involved in the step case, except that extra operators are required to keep the inductive hypothesis in line with the step goal.

Thus, the hypothesis, 13. $e = u \mid mem(e, z1)$, is decomposed further by the operator, `remove_union_hypothesis`, which splits the hypothesis into the two hypotheses, $e = u$ and $mem(e, z1)$. This results in two identical sub-goals

$$\gg\ e = a \mid mem(e, u \cdot x1)$$

each with one of the above two hypotheses. The proof of the right sub-goal is continued in the next section.

In NuPRL, only one of the terms connected by the union connective, |, need be proved [4]. The proof of the left sub-goal involves the application of the operator, `remove_left_union`, leaving the *rhs* of the union term to be proved, followed by the `unfold` operator, which results in the sub-goal

$$\gg\ e = u \mid mem(e, x1)$$

Applying the operator, `remove_right_union` produces a sub-goal which matches one of the previous hypotheses, 14, such that the `hypothesis` operator terminates this branch of the step case proof.

[4] In a sense, an expression containing the union connective acts like a disjunction of terms.

$1-7$
$\gg\ I(a, u \cdot x1, u \cdot z1)$

by parm_tac

$1-7$
8. e
$\gg\ mem(e, u \cdot z1) \Leftrightarrow e = a \mid mem(e, u \cdot x1)$

by remove_universal_hypothesis 7

$1-8$
9. $mem(e, z1) \Leftrightarrow e = a \mid mem(e, x1)$
$\gg\ mem(e, u \cdot z1) \Leftrightarrow e = a \mid mem(e, u \cdot x1)$

by remove_equivalence_hypothesis 9

$1-9$
10. $mem(e, z1) \rightarrow e = a \mid mem(e, x1)$
11. $e = a \mid mem(e, x1) \rightarrow mem(e, z1)$
$\gg\ mem(e, u \cdot z1) \Leftrightarrow e = a \mid mem(e, u \cdot x1)$

by remove_equivalence

$1-11$
$\gg\ mem(e, u \cdot z1) \rightarrow$
$\quad e = a \mid mem(e, u \cdot x1)$

by unfold

$1-11$
$\gg\ e = u \mid mem(e, z1) \rightarrow$
$\quad e = a \mid mem(e, u \cdot x1)$

by remove_implication

$1-11$
12. $e = u \mid mem(e, z1)$
$\gg\ e = a \mid mem(e, u \cdot x1)$

•

$1-11$
$\gg\ e = a \mid mem(e, u \cdot x1) \rightarrow$
$\quad mem(e, u \cdot z1)$

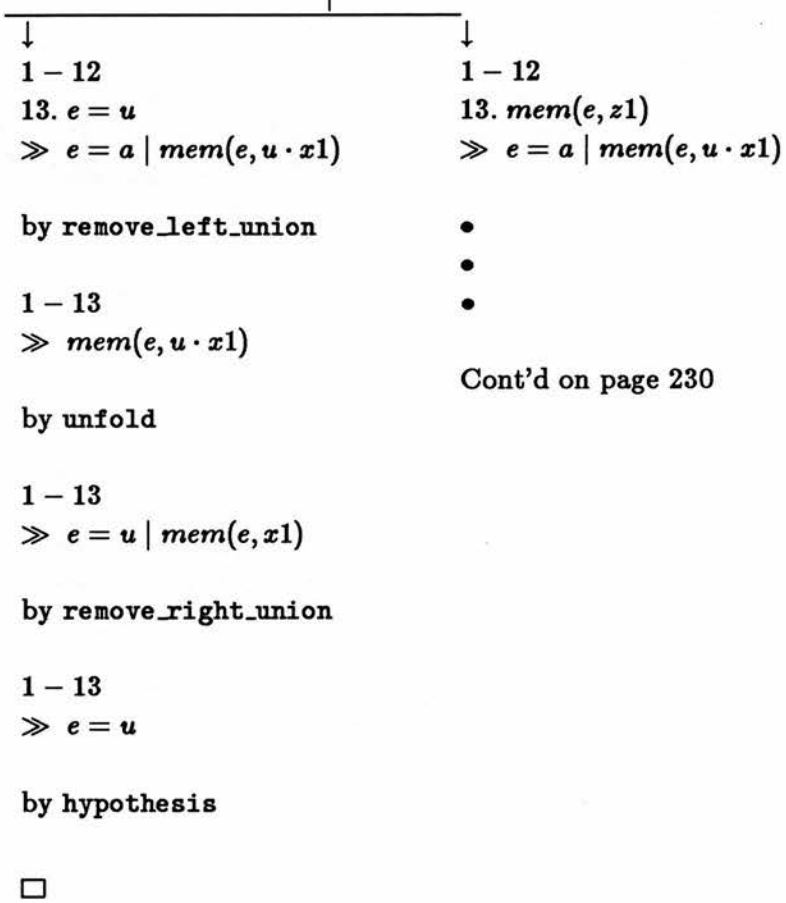
•
•
•

Not Continued

Cont'd on page 228

$1 - 12$
$\gg\ e = a \mid mem(e, u \cdot x1)$

by remove_union_hypothesis 12



A.2.3 Step Case Proof Completed

Refer to page 230. The next operator, `remove_implication_hypothesis` takes part of the inductive hypothesis, 11, and removes the implication connective by adding the *lhs* to the list of hypotheses for the sub-goal, resulting in the following goal to prove

$$15.\ e = a \mid mem(e, x1)$$
$$\gg\ e = a \mid mem(e, u \cdot x1)$$

However, for this operation to be considered valid, the *rhs* of the implication must also be proved. This happens to be trivial since the sub-goal, $mem(e, z1)$, already belongs to the list of hypotheses, due to the hypothesis, 13. Thus the `hypothesis` operator terminates this branch of the step case proof.

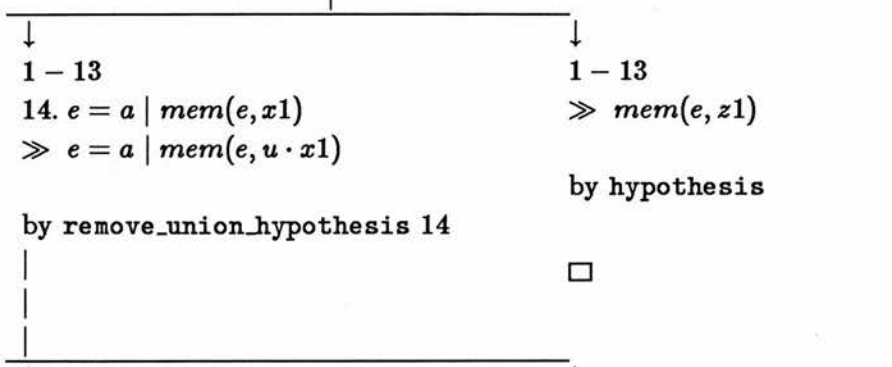
In the remaining branch of the proof, the next operator, `remove_union_hypothesis`, splits hypothesis 15 in two further hypotheses, $mem(e, x1)$ and $e = a$. This results in two identical sub-goals to prove, each with one of the above new hypotheses.

Both branches of the proof tree are terminated by the application of some combination of the following operators:

`remove_left_union`
`remove_right_union`
`unfold` and
`hypothesis`

$1 - 13$
$\gg\ e = a \mid mem(e, u \cdot x1)$

by remove_implication_hypothesis 11



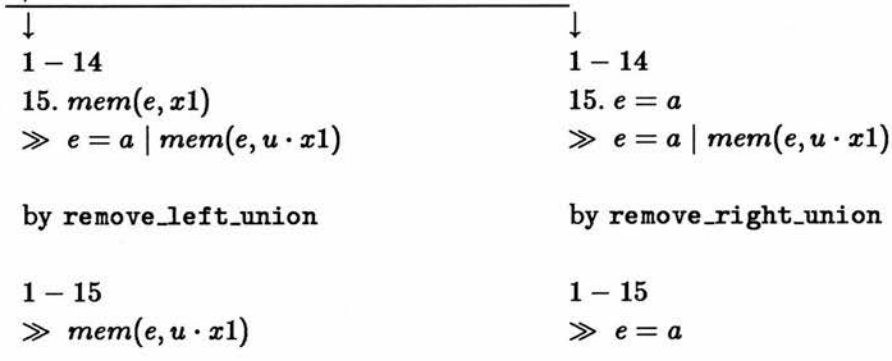
$1 - 13$
14. $e = a \mid mem(e, x1)$
$\gg\ e = a \mid mem(e, u \cdot x1)$

by remove_union_hypothesis 14

$1 - 13$
$\gg\ mem(e, z1)$

by hypothesis

□



$1 - 14$
15. $mem(e, x1)$
$\gg\ e = a \mid mem(e, u \cdot x1)$

by remove_left_union

$1 - 15$
$\gg\ mem(e, u \cdot x1)$

by unfold

$1 - 15$
$\gg\ e = u \mid mem(e, x1)$

by remove_left_union
$1 - 15$
$\gg\ mem(e, x1)$

by hypothesis

□

$1 - 14$
15. $e = a$
$\gg\ e = a \mid mem(e, u \cdot x1)$

by remove_right_union

$1 - 15$
$\gg\ e = a$

by hypothesis

□

Appendix B

Extended-EBL Program Listing

This appendix contains a listing of the main code involved in the Extended-EBL program, described within this thesis.

```
%       EXTENDED-EBL                                    12/12/87
%
% Extended-EBL Program incorporating the best features of the
% Precondition Analysis and Back Propagation techniques.

:-      [identification,                % Call to files containing
         precondition,                  % main prolog code
         recon_data,
         mygen,string],
        compile([library(basics),
                 library(lists),
                 library(sets),
                 library(flatten)]).

:-      dynamic proof_plan/2.

learn_proof_plan(Step,Back) :-
        operator_identification(Step),
        precondition_analysis(Step,Back,RT,RS).
```

```
%        OPERATOR IDENTIFICATION                    04/03/87
%
% Identify the operators involved in the problem and provides
% a skeleton structure for each operator which is expanded
% later on during the extended precondition analysis stage.
%
% Each structure is of the form:
%    operator_structure(Name_of_Operator,
%                       Object_level_state_before_operator,
%                       List_of_Preconditions_for_operator,
%                       List_of_Effects_for_operator).
%
% The list of effects for each operator comprises a list
% structure of the form:
%  effects(Object_level_state_after_operator,
%          Postconds_of_operator_matching_object_level_state)

:-      [find_operator].

operator_identification(Step) :-      % Top level call
   object_level_proof(                % Find consequences of Step
        Step,_,Consequences),         % Find operators and assert
   find_ops_and_assert_structure(     % structure for each one
        Step,Consequences).

find_ops_and_assert_structure(        % If consequences empty
        _,[]) :- !.                   % then do not find operators
                                      % nor generate structure

find_ops_and_assert_structure(
        Step,Consequences) :-
   find_operator(                     % Find operator and return
        Step,Consequences,            % pre- and postconditions
        Operator,Preconditions,
        Postconditions),
```

```
   identify_rest_of_operators(        % Identify rest of operators
        Consequences,                 % and return list of effects
        Postconditions,
        Effects_structure),
   asserta(                           % Assert structure
        operator_structure(
           Operator,Step,
           Preconditions,
           Effects_structure)).


identify_rest_of_operators(           % If consequences and
        [],[],[]) :- !.               % postconditions empty
                                      % return empty effects list

identify_rest_of_operators(           % If postconditions empty
        [Step|Rest_of_consequences],  % because operator unknown
        [],                           % then return list of effects
        [effects(Step,[])             % with empty postconditions
        |Rest_of_effects_structure]) :-
   operator_identification(Step),
   identify_rest_of_operators(
        Rest_of_consequences,[],
        Rest_of_effects_structure).

identify_rest_of_operators(           % Otherwise,
        [Step|Rest_of_consequences],  % return list of effects
        [Postconditions               % with object-level states
        |Rest_of_postconditions],     % and relevant postconditions
        [effects(Step,Postconditions)
        |Rest_of_effects_structure]) :-
   operator_identification(Step),
   identify_rest_of_operators(
        Rest_of_consequences,
        Rest_of_postconditions,
```

```
        Rest_of_effects_structure).



%       FIND OPERATOR                                   04/03/87
%
% Finds a operator whose preconditions and postconditions
% match the object-level states before and after the operator,
% ie Step and Consequences.
% Return the name of the operator, together with its
% preconditions and postconditions.
%
% Otherwise if no existing operator matches these states,
% then generate a skeleton operator with empty specifications.
%
%
:-      [check_preconditions,
         check_postconditions].

find_operator(                          % Top level call
        Step,Consequences,
        Operator,Preconditions,
        Postconditions) :-
   operator(Operator,Preconditions,     % Search for operator
        Postconditions,_),
   check_preconditions(                 % Check preconditions
        Preconditions,Step),            % against step formula.
   check_postconditions(                % Check postconditions
        Postconditions,                 % against consequences
        Step,Consequences).             % of step.

find_operator(_,_,Name,[],[]) :-        % If operator not known
   new_operator_name(Name),             % find distinct name
   assertz(                             % assert operator with
       operator(Name,[],[],ops)).       % empty specifications
```

```
new_operator_name(Name) :-          % code for generating
   retract(                         % new operator name.
     operator_count(Operator_count)),
   restore_mygensym_counts(
        operator,Operator_count),
   mygensym(operator,t,Name),
   get_mygensym_counts(
        operator,Rootcounts),
   assert(operator_count(Rootcounts)).



%       PRECONDITION ANALYSIS                    11/11/87
%
% Apply precondition analysis process from initial step,
% Step, through the entire operator structure.  For each
% level of the problem return the back propagated
% preconditions.

:-      [revise_rest,
         revise_operators_preconds,
         revise_operator_postconds,
         match_sets,
         find_back_propagated].

precondition_analysis(Step,         % Top level precondition
        Bp_preconditions,           % analysis
        Another_operator,
        Another_state)  :-
    operator_structure(             % Find operator structure
        Operator,Step,              % which matches Step
        Preconditions,
        Effects_structure),
    revise_rest_of_op_structure(    % Return Preconditions for
```

```
        Effects_structure,          % the next stages, operators
        Preconds_for_next_stages,   % and states for determining
        Ops_with_missing_preconds,  % missing preconditions
        States),
    revise_ops_with_missing_pre(    % Revise operators with
        Effects_structure,          % missing preconditions
        Ops_with_missing_preconds,
        States,
        Preconds_for_next_stages,
        Revised_preconds),
    match_sets(                     % Match postconditions with
        Revised_preconds,           % preconditions for next
        Step,Effects_structure,     % stages - return matched
        Revised_effects_structure,  % and unmatched preconds,
        Postconditions,             % also postconditions
        Set_of_Matched_preconds,    % and revised effects
        Set_of_Unmatched_preconds),
    revise_op_with_missing_post(    % revise current operator
        Operator,Step,              % with missing postconditions
        Preconditions,              % if required
        Postconditions,
        Another_operator,
        Another_state),
    find_back_propagated(           % Find Back propagated
        Set_of_Matched_preconds,    % preconditions for
        Set_of_Unmatched_preconds,  % current level of proof
        Preconditions,
        Revised_effects_structure,
        Bp_preconditions).

precondition_analysis(              % If no more operators return
        Step,[],[],[]) :-           % [] for back propagated
    \+(operator_structure(          % preconditions
        _,Step,_,_)),!.
```

```
%       REVISE REST OF OPERATOR STRUCTURE          04/04/87
%
% Apply precondition analysis to the rest of the operator
% structure


revise_rest_of_op_structure(          % Stop, if no more effects
        [],[],[],[]) :- !.

revise_rest_of_op_structure(          % Recursively call
        [effects(Step,_)              % precondition analysis
        |Rest_of_effects_structure], % on effects structure
        [Preconds_for_next_stage
        |Rest_of_preconditions],
        List_of_operators,
        List_of_states) :-
    precondition_analysis(Step,       % Call precondition analysis
        Preconds_for_next_stage,      % return next preconditions
        Operators,States),
    revise_rest_of_op_structure(      % Expand effects tree
        Rest_of_effects_structure,    % structure
        Rest_of_preconditions,
        More_operators,More_states),
    append(Operators,More_operators,
        List_of_operators),
    append(States,More_states,
        List_of_states).




%       REVISE OPERATORS WITH MISSING PRECONDITIONS    11/11/87
%
% Find the preconditions of missing operators and revise the
% relevant operators to reflect these newly determined
% preconditions
```

```
:-      [find_preconditions].

revise_ops_with_missing_pre(_,          % Stop, if [] Operators and
        [],[],P,P,) :- !.               % states

revise_ops_with_missing_pre(            % Revise operators with
        Effects_structure,              % missing preconditions
        [Op_with_missing_pre
        |Rest_of_operators],
        [State_for_determining_pre
        |Rest_of_states],
        Preconds_for_next_stages,
        Revised_preconds) :-
    find_preconditions_for_state(       % Find the missing preconds
        Effects_structure,
        State_for_determining_pre,
        Preconds_for_next_stages,
        Revised_preconds1,
        Preconditions),
    retract(operator(
        Op_with_missing_pre,[],
        List_of_postconditions,Ops)),
    assertz(operator(                   % Revise the relevant operator
        Op_with_missing_pre,
        Preconditions,
        List_of_postconditions,
        Ops)),
    revise_ops_with_missing_pre(        % Find preconditions for
        Effects_structure,              % rest of missing operators
        Rest_of_operators,
        Rest_of_states,
        Revised_preconds1,
        Revised_preconds).
```

```
revise_ops_with_missing_pre(          %
        Effects_structure,
        [Operator|Rest_of_operators],
        [State|Rest_of_states],
        Preconds_for_next_stages,
        Revised_preconds) :-
    revise_ops_with_missing_pre(      %
        Effects_structure,
        Rest_of_operators,
        Rest_of_states,
        Preconds_for_next_stages,
        Revised_preconds).



% MATCH SETS OF POSTCONDITIONS WITH PRECONDITIONS
%
% Match the postconditions of the current operator with
% the preconditions for the next stages of the problem.
% Return the matched and unmatched preconditions.

:- [match_postconditions,
         match_preconditions,
         find_equivalent_postconds].

match_sets([],[],[],[],_) :- !.       % Stop, if no more effects
                                      % return [] matched and
                                      % unmatched preconditions

match_sets(                           % Match sets of postcondition
        [Pre|Rest_of_pre],            % from each effect with
        Previous_step,                % preconditions
        [effects(Step,Post)
        |Rest_of_effects_structure],
        [effects(Step,Post)
```

```
        |Rest_of_revised_effects],
        [Post|Rest_of_post],
        [Matched_preconditions
        |Rest_of_matched],
        [Unmatched_preconditions
        |Rest_of_unmatched]) :-

    match_post_with_pre(            % Match postconditions for
        Post,Prec,                  % current effects -
        Matched_preconditions,      % return matched and
        Unmatched_preconditions),   % unmatched preconditions

    match_sets(                     % Recursively call other
        Rest_of_pre,                % sets of postconditions
        Previous_step,
        Rest_of_effects_structure,
        Rest_of_revised_effects,
        Rest_of_post,
        Rest_of_matched,
        Rest_of_unmatched).

match_sets([Pre|Rest_of_pre],       % If postconditions []
        Previous_step,              % because of missing operator
        [effects(Step,[])
        |Rest_of_effects_structure],
        [effects(Step,Post)
        |Rest_of_revised_effects],
        [Post|Rest_of_post],
        [Matched_preconditions
        |Rest_of_matched],
        [Unmatched_preconditions
        |Rest_of_unmatched]) :-
    match_preconditions(            % Match preconditions with
        Pre,Previous_step,          % object-level state of
        Matched_preconditions,      % previous step
```

```
        Unmatched_preconditions),
    find_equiv_post(
        Matched_preconditions,
        Post),
    match_sets(                         % Recursively call other
        Rest_of_pre,                    % sets of postconditions
        Previous_step,
        Rest_of_effects_structure,
        Rest_of_revised_effects,
        Rest_of_post,
        Rest_of_matched,
        Rest_of_unmatched).



%       REVISE OPERATOR WITH MISSING POSTCONDITIONS
%
% Revise the missing operator with the newly determined
% postconditions

revise_op_with_missing_post(
        Op,Step,[],
        Postconditions,
        [Op],[Step]) :-
    retract(operator(Op,[],[],Ops)),
    assertz(operator(Operator,[],
        Postconditions,Ops)).

revise_op_with_missing_post(
        _,_,_,[],[]) :- !.



% FIND BACK PROPAGATED PRECONDITIONS        23/04/87
%
```

```
% Find the back propagated preconditions from each of the
% branches of the problem and remove any duplicate
% preconditions that arise.

:- [find_candidate].

find_back_propagated(                  % Return Back Propagated
        Set_of_Matched,                % preconditions
        Set_of_Unmatched,
        Precond_list,
        Effects_structure,
        Back_propagated) :-
    find_candidate_bp(                 % Find back propagated
        Set_of_Matched,                % preconditions from all
        Set_of_Unmatched,              % effects including
        Precond_list,                  % duplicates
        Effects_structure,
        Candidate_bp),
    flatten(Candidate_bp,              % Flatten sets of bp preconds
        Flattened_cands),
    remove_dups(Flattened_cands,       % Remove duplicates
        Back_propagated).
```

Appendix C

Output Trace for the Insert Proof

C.1 Introduction

This appendix contains the output trace of the Extended-EBL program, when the partial insert proof, described in chapter 6, is the input. The partial proof provided to the Extended-EBL is shown in figure C–1.

Remember that the Extended-EBL program traverses through the proof structure in a depth-first manner, choosing the left-most sub-tree at each branch of the proof.

At each step of the proof, the program records the structure of the sub-plan so far together with the preconditions for that sub-plan. Note that the Extended-EBL program does not have any fancy graphic facilities for displaying the structure of the learned plan. However, for the sake of clarity the slot denoted *structure of the plan* comprises a graphical representation of this structure, but is NOT the actual output of the program.

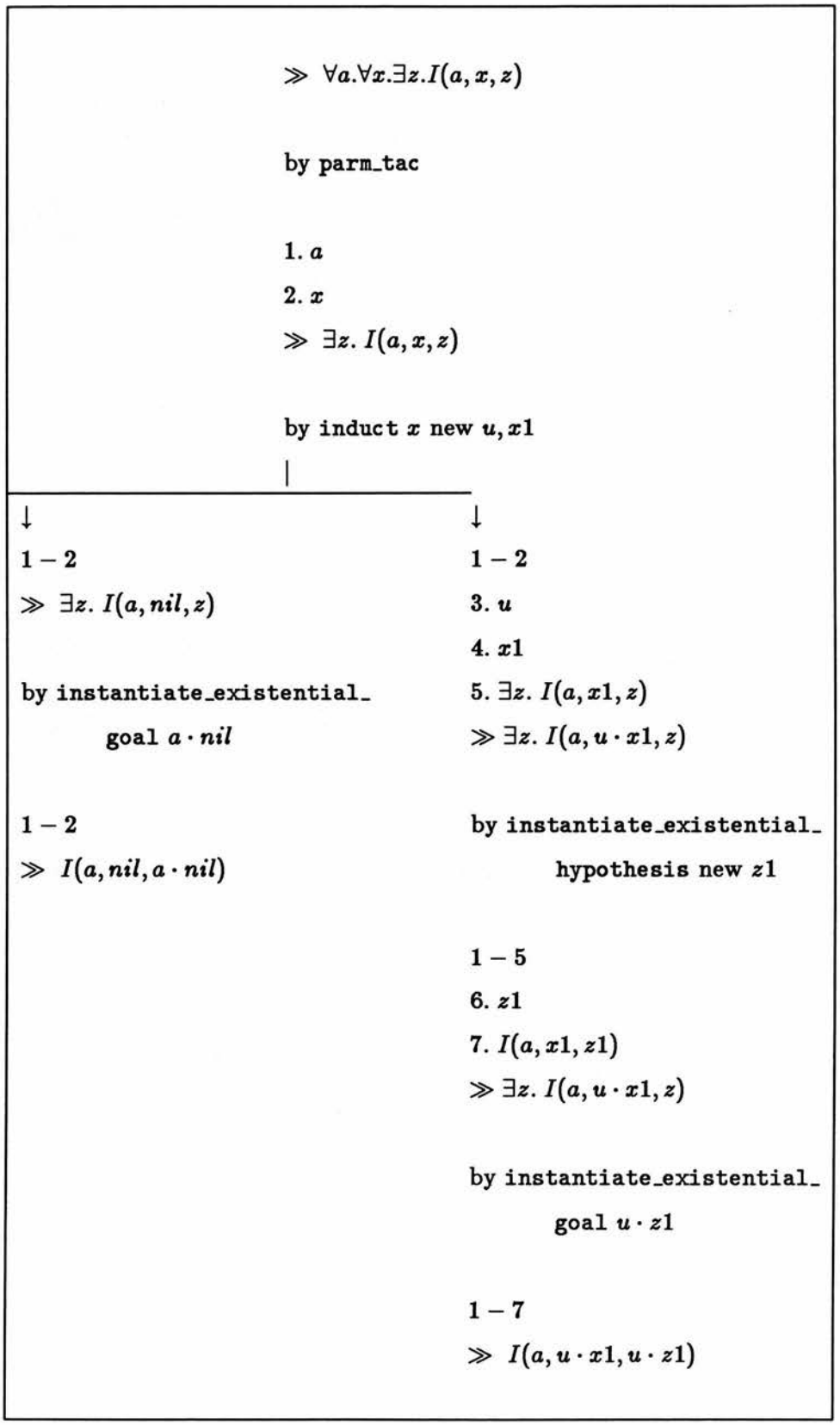
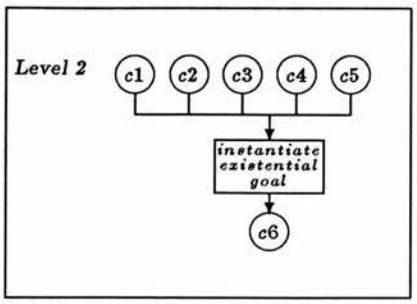


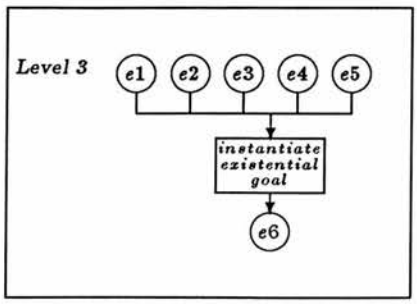
Figure C–1: Representation of a partial proof tree

C.2 Output Trace

```
**********************************************************************
Sub-Plan for proving:
        1-2
        >> exists z. I(a,nil,z)

Matched preconditions:
        There are no matched preconditions!

Unmatched preconditions:
There are no unmatched preconditions!

Back propagated preconditions:
There are no back propagated preconditions!

Preconditions for the sub-plan:
        decompose(exists z. I(a,nil,z), exists z, I(a,nil,z))    c1
        existential([z], exists z. I(a,nil,z))                   c2
        contain(I(a,nil,z), [z])                                 c3
        hypothesis(a, [a,x])
        hypothesis(nil,[a,x])                                    c4
        goal(exists z. I(a,nil,z))                               c5

Structure of the sub-plan:
```



```
**********************************************************************
Sub-Plan for proving:
        1-8
        >> exists z. I(a,u.x1,z)

Matched preconditions:
        There are no matched preconditions!

Unmatched preconditions:
There are no unmatched preconditions!

Back propagated preconditions:
There are no back propagated preconditions!

Preconditions for the sub-plan:
        decompose(exists z. I(a,u.x1,z), exists z, I(a,u.x1,z))  e1
        existential([z], exists z. I(a,u.x1,z))                  e2
        contain(I(a,u.x1,z), [z])                                e3
        goal(exists z. I(a,u.x1,z))                              e4
        hypothesis(u,
                  [a,x,u,x1,exists z. I(a,x1,z),z1,I(a,x1,z1)])  e5b
        hypothesis(z1,
                  [a,x,u,x1,exists z. I(a,x1,z),z1,I(a,x1,z1)])  e5a

Structure of the sub-plan:
```



```
**********************************************************************
Sub-Plan for proving:
        1-5
        >> exists z. I(a,u.x1,z)

Matched preconditions:
        hypothesis(z1,
                [a,x,u,x1,exists z. I(a,x1,z),z1,I(a,x1,z1)])  e5a

  matches with effect:

        include([z1,I(a,x1,z1)],
              [a,x,u,x1,exists z. I(a,x1,z)]),
              [a,x,u,x1,exists z. I(a,x1,z),z1,I(a,x1,z1)]      d6

  as a result of the inference:

        include(Hyp, Old_hyplist, New_hyplist)
        --> hypothesis(Hyp, New_hyplist)                        1

Unmatched preconditions:
        decompose(exists z. I(a,u.x1,z), exists z, I(a,u.x1,z))  e1
        existential([z], exists z. I(a,u.x1,z))                  e2
        contain(I(a,u.x1,z), [z])                                e3
        goal(exists z. I(a,u.x1,z))                              e4
        hypothesis(u,
                [a,x,u,x1,exists z. I(a,x1,z),z1,I(a,x1,z1)])  e5b

Back propagated preconditions:
        decompose(exists z. I(a,u.x1,z), exists z, I(a,u.x1,z))  e1
        existential([z], exists z. I(a,u.x1,z))                  e2
        contain(I(a,u.x1,z), [z])                                e3
        goal(exists z. I(a,u.x1,z))                              e4
        hypotheses(u, [a,x,u,x1,exists z. I(a,x1,z)])            e5'
```

Back propagating over state transition requires:

replacing occurrences of new hypothesis list:

[a,x,u,x1,exists z. I(a,x1,z),z1,I(a,x1,z1)]

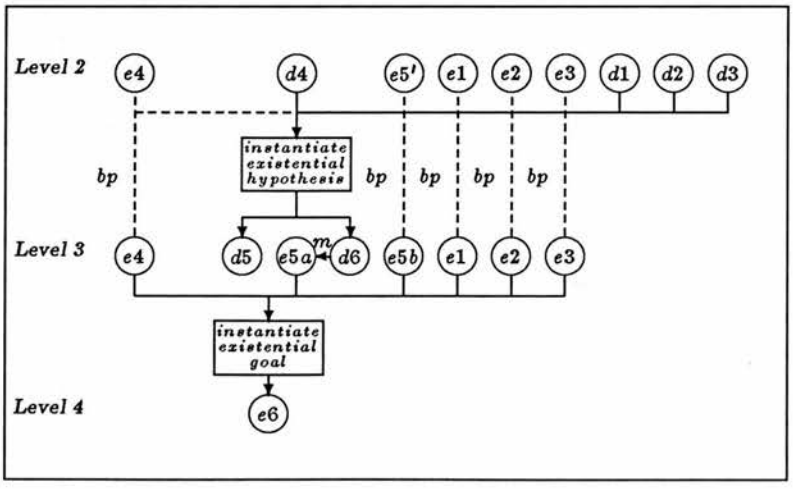
by the old hypothesis list:

[a,x,u,x1,exists z. I(a,x1,z)]

Preconditions for the sub-plan:

decompose(exists z. I(a,x1,z), exists z, I(a,x1,z))	d1
existential([z], exists z. I(a,x1,z))	d2
contain(I(a,x1,z), [z])	d3
hypothesis(exists z. I(a,x1,z), [a,x,u,x1,exists z. I(a,x1,z)])	d4
decompose(exists z. I(a,u.x1,z), exists z, I(a,u.x1,z))	e1
existential([z], exists z. I(a,u.x1,z))	e2
contain(I(a,u.x1,z), [z])	e3
goal(exists z. I(a,u.x1,z))	e4
hypothesis(u, [a,x,u,x1,exists z. I(a,x1,z)])	e5'

Structure of the sub-plan:



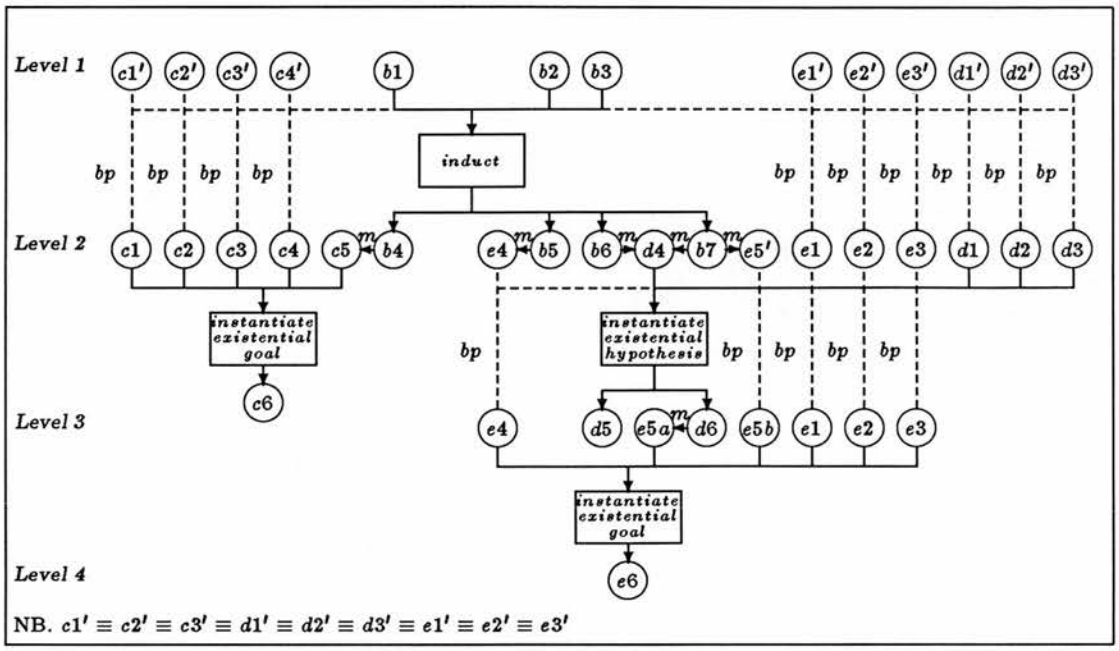
****

```
Sub-Plan for proving:
        1-2
        >> exists z. I(a,x,z)

Matched preconditions:
        goal(exists z. I(a,nil,z))                                    c5

  matches with effect:

        replace_all(x,nil,exists z.I(a,x,z),exists z.I(a,nil,z)) b4

  as a result of the inference:

        goal(Old_state) &
        base(Old_term,New_term,Ind_scheme) &
        replace_all(Old_term,New_term,Old_state,New_state)
        --> goal(New_state)                                           2

-------
        goal(exists z. I(a,u.x1,z))                                   e4

  matches with effect:

        replace_all(x,u.x1,exists z.I(a,x,z),exists z.I(a,u.x1,z)  b5

  as a result of the inference:

        goal(Old_state) &
        step(Old_term,New_term,Ind_scheme) &
        replace_all(Old_term,New_term,Old_state,New_state)
        --> goal(New_state)                                           3

-------
      hypothesis(exists z. I(a,x1,z))                                 d4
```

```
  matches with effect:

        replace_all(x,x1,exist z.I(a,x,z),exists z. I(a,x1,z)    b6

  as a result of the inference:

        goal(Old_state) &
        ind_hypothesis(Old_term,New_term,Ind_scheme) &
        replace_all(Old_term,New_term,Old_state,New_state)
        --> hypothesis(New_state)                                 4

-------
      hypothesis(exists z. I(a,x1,z))                             d4

  matches with effect:

        include([u,x1,exists z.I(a,x1,z)],
                [a,x],
                [a,x,u,x1,exists z.I(a,x1,z)])                    b7

  as a result of the inference:

        include(Hyp, Old_hyplist, New_hyplist)
        --> hypothesis(Hyp, New_hyplist)                          1

--------
      hypothesis(u, [a,x,u,x1,exists z.I(a,x1,z)])                e5'

  matches with effect:

        include([u,x1,exists z.I(a,x1,z)],
                [a,x],
                [a,x,u,x1,exists z.I(a,x1,z)])                    b7

  as a result of the inference:
```

```
        include(Hyp, Old_hyplist, New_hyplist)
        --> hypothesis(Hyp, New_hyplist)                                  1

Unmatched preconditions:
        decompose(exists z. I(a,nil,z), exists z, I(a,nil,z))     c1
        existential([z], exists z. I(a,nil,z))                    c2
        contain(I(a,nil,z), [z])                                  c3
        hypothesis(a, [a,x])
        hypothesis(nil,[a,x])                                     c4
        decompose(exists z. I(a,x1,z), exists z, I(a,x1,z))       d1
        existential([z], exists z. I(a,x1,z))                     d2
        contain(I(a,x1,z), [z])                                   d3
        decompose(exists z. I(a,u.x1,z), exists z, I(a,u.x1,z))   e1
        existential([z], exists z. I(a,u.x1,z))                   e2
        contain(I(a,u.x1,z), [z])                                 e3

Back propagated preconditions:
        decompose(exists z. I(a,x,z), exists z, I(a,x,z))         c1'
        existential([z], exists z. I(a,x,z))                      c2'
        contain(I(a,x,z), [z])                                    c3'
        hypothesis(a, [a,x])
        hypothesis(x, [a,x])                                      c4'

Back propagating over state transition requires:
        replacing occurrences of new term:
        nil
        by the old term:
        x
-------
        replacing occurrences of new term:
        u.x1
        by the old term:
        x
-------
```

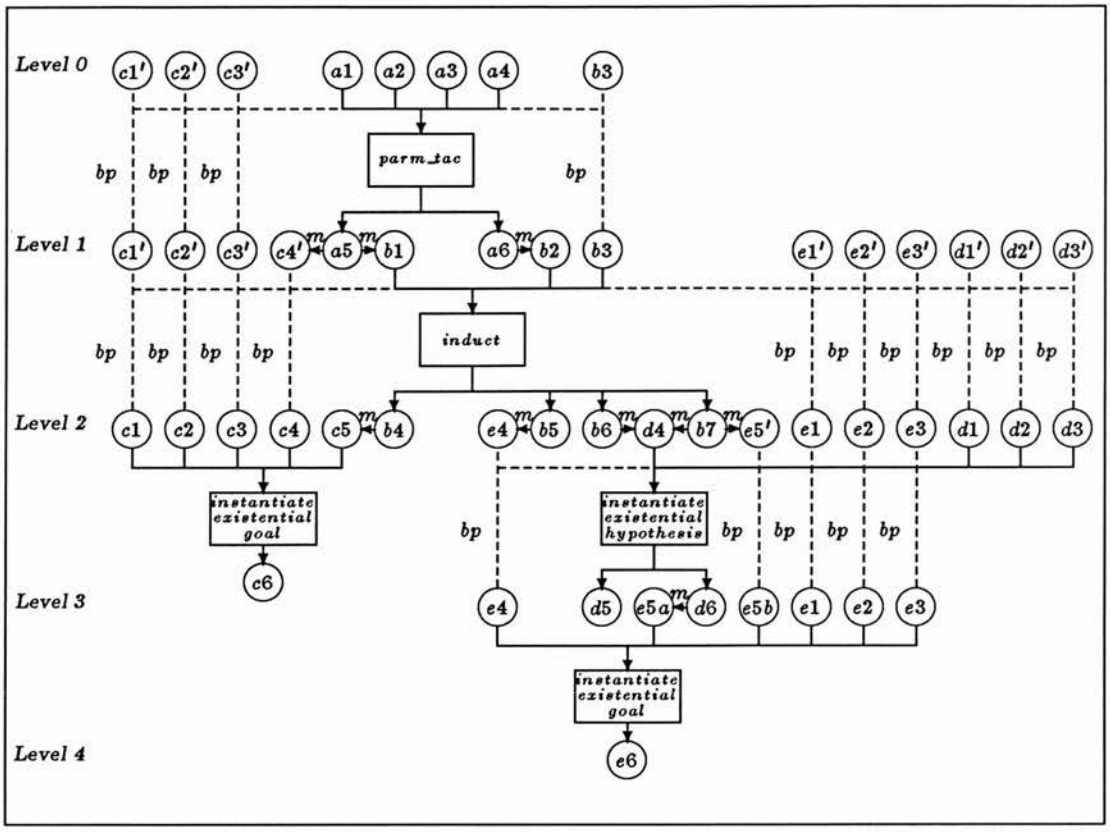
```
        replacing occurrences of new term:
        x1
        by the old term:
        x
-------
        replacing the occurrence of the new hypothesis list:
        [a,x,u,x1,exists z.I(a,x1,z)])
        by the old hypothesis list:
        [a,x]

Preconditions for the sub-plan:
        hypothesis(x, [a,x])                                        b1
        goal(exists z. I(a,x,z))                                    b2
        contain(exists z.I(a,x,z), [x])                             b3
        decompose(exists z. I(a,x,z), exists z, I(a,x,z))           c1'
        existential([z], exists z. I(a,x,z))                        c2'
        contain(I(a,x,z), [z])                                      c3'
        hypothesis(a, [a,x])
        hypothesis(x, [a,x])                                        c4'


Structure of the sub-plan:
```




```
**********************************************************************
Overall Plan for proving:
        >> forall a. forall x. exists z. I(a,x,z)

Matched Preconditions:
        hypothesis(x,[a,x])                                        b1

  matches the effect:
        include([a,x],[],[a,x])                                    a5

  as a result of the inference:

        include(Hyp, Old_hyplist, New_hyplist)
        --> hypothesis(Hyp, New_hyplist)                           1

--------
```

```
        hypothesis(a,[a,x])                                          c4'

  matches the effect:
        include([a,x],[],[a,x])                                      a5

  as a result of the inference:

        include(Hyp, Old_hyplist, New_hyplist)
        --> hypothesis(Hyp, New_hyplist)                             1

--------
        goal(exists z. I(a,x,z)                                      b2

  matches the effect:
        remove_quantifier(
                forall a. forall x. exists z. I(a,x,z),
                [forall a, forall x],
                exists z. I(a,x,z))                                  a6

  as a result of the inference:
        goal(Old_state) &
        remove_quantifier(Old_state,Quantifiers,New_state)
        -->  goal(New_state)                                         6

Unmatched preconditions:
        contain(exists z.I(a,x,z), [x])                              b3
        decompose(exists z. I(a,x,z), exists z, I(a,x,z))            c1'
        existential([z], exists z. I(a,x,z))                        c2'
        contain(I(a,x,z), [z])                                       c3'

Back propagated preconditions:
        contain(exists z.I(a,x,z), [x])                              b3
        decompose(exists z. I(a,x,z), exists z, I(a,x,z))            c1'
        existential([z], exists z. I(a,x,z))                        c2'
        contain(I(a,x,z), [z])                                       c3'
```

```
Back propagating over state transition requires:
        replacing the occurrence of the new hypothesis list:
        [a,x]
        by the old hypothesis list:
        []

Preconditions for the sub-plan:
        contain(exists z. I(a,x,z), [a,x])                          a1
        universal([a,x],
                  forall a. forall x. exists z. I(a,x,z))           a2
        decompose(forall a. forall x. exists z. I(a,x,z),
                  [forall a, forall x],
                  exists z. I(a,x,z))                               a3
        goal(forall a. forall x. exists z. I(a,x,z))                a4
        contain(exists z.I(a,x,z), [x])                             b3
        decompose(exists z. I(a,x,z), exists z, I(a,x,z))           c1'
        existential([z], exists z. I(a,x,z))                       c2'
        contain(I(a,x,z), [z])                                      c3'


Structure of the sub-plan:
```


Level 0
Level 1
Level 2
Level 3
Level 4
parm_tac
induct
instantiate existential goal
instantiate existential hypothesis
instantiate existential goal
bp
m